PC professionell
Arbeitsplatzrechner in Ausbildung und Praxis

A.T. Schreiner
G. Friedman

Compiler bauen mit UNIX

Eine Einführung

HANSER

Schreiner/Friedman
Compiler bauen mit UNIX

PC professionell
Arbeitsplatzrechner in Ausbildung und Praxis

Herausgeber:
Prof. Dr.-Ing. Werner Heinzel, Fulda

PC professionell zeigt dem Anwender von Personal Computern den Aufbau, die Programmierung und die verschiedenen Anwendungsmöglichkeiten seines „persönlichen Rechners am Arbeitsplatz" und führt ihn am Beispiel aktueller PC-Systeme oder mit der Darstellung und Diskussion von Fallbeispielen in die betriebliche Praxis derartiger Geräte ein.

Personalcomputer werden in den verschiedensten Anwendungsbereichen eingesetzt:

- in Klein- und Mittelbetrieben
- als persönliches Arbeitsmittel in Fachabteilungen von Großbetrieben
- als Ausbildungsmittel in Schulen, Hochschulen und Fortbildungsstätten
- im privaten Bereich

PC professionell wendet sich an diese verschiedenen Benutzer des Personal Computers.

Aufgrund des didaktischen Aufbaus und der eingearbeiteten Praxisbeispiele eignen sich die Bände zum Selbststudium.

Die Fachbuchreihe weist die folgenden thematischen Schwerpunkte auf:

Grundlagen der Hard- und Softwaretechnologie von Personal Computern,
zum Beispiel

- Systemstrukturen
- Betriebssysteme
- Programmiersprachen
- Softwarewerkzeuge

Realisierte Systeme und deren Anwendungen,
zum Beispiel

- Benutzeranleitung für bestimmte PC-Systeme
- Programmierung typischer Systeme
- Anwendungen, Fallbeispiele, Softwaresammlungen
- Softwarelösungen für bestimmte Branchen bzw. Anwendungsgebiete

Carl Hanser Verlag München Wien

A. T. Schreiner/H. G. Friedman

Compiler bauen mit UNIX

Eine Einführung

Die deutsche Ausgabe besorgten
Prof. Dr. A. T. Schreiner und Dr. Ernst Janich

Carl Hanser Verlag München Wien

Prof. Dr. Axel T. Schreiner
Sektion Informatik der Universität Ulm

Prof. H. George Friedman, Jr.
Department of Computer Science University
of Illinois at Urbana-Champaign

Titel der amerikanischen Ausgabe:
Introduction to Compiler Construction with UNIX
© 1985 Prentice-Hall, Inc.

UNIX ist ein eingetragenes Warenzeichen der Bell Laboratories. Das UNIX Betriebssystem wird in Lizenz vergeben durch AT&T International

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Schreiner, Axel-Tobias:
Compiler bauen mit UNIX : e. Einf. / A. T.
Schreiner ; H. G. Friedman. - München ; Wien :
Hanser, 1985.
(PC professionell)
ISBN 3-446-14359-9

NE: Friedman, Harry G.:

Dieses Werk ist urheberrechtlich geschützt.
Alle Rechte, auch die der Übersetzung, des Nachdrucks und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

©1985 Carl Hanser Verlag München Wien

Satz und Druck: Druckerei Appl, Wemding
Buchbinderische Verarbeitung: Sellier, Freising
Umschlagentwurf: Carl-Alfred Loipersberger
Printed in Germany

Für Carol und Claudia,
die's mit uns aufnehmen

Inhaltsverzeichnis

Kapitel 0: Einführung 1

Kapitel 1: Sprachdefinition 5
1.1 Zweck 5
1.2 Formale Aspekte 7
1.3 Zweckmäßige Grammatiken 11
1.4 Überprüfung einer Grammatik 14
1.5 Fallstricke 18
1.6 Beispiel 22
1.7 Zur Typographie 28
1.8 Aufgaben 29

Kapitel 2: Worterkennung 31
2.1 Zweck 31
2.2 Werkzeuge zur Definition 31
2.3 Muster 34
2.4 Mehrdeutigkeit als Tugend 38
2.5 „lex"-Programme 39
2.6 Fehlersuche bei der lexikalischen Analyse 44
2.7 Beispiel 45
2.8 Aufgaben 49

Kapitel 3: Spracherkennung 51
3.1 Generierung eines Parsers 51
3.2 Beispiel 55
3.3 Hilfsfunktionen 57
3.4 Fehlersuche im Parser 62
3.5 Bezeichner 69
3.6 Typdefinition für den Wert-Stack 72
3.7 Beispiel 73
3.8 Aufgaben 84

Kapitel 4: Fehlerbehandlung 87
4.1 Problemstellung 87
4.2 Elementare Techniken 88
4.3 Plazierung der „error"-Symbole 94
4.4 Plazierung der „yyerrok"-Aktionen 99
4.5 Beispiel 99
4.6 Aufgaben 108

Kapitel 5: Semantikprüfung 109
5.1 Problemstellung 109
5.2 Symboltabellen 112

5.3 Beispiel . . . 113
5.4 Typdefinition für den Wert-Stack . . . 134
5.5 Aufgaben . . . 136

Kapitel 6: Speicherverwaltung . . . 137
6.1 Grundbegriffe . . . 137
6.2 Beispiel . . . 141
6.3 Aufgaben . . . 149

Kapitel 7: Code-Generierung . . . 151
7.1 Grundbegriffe . . . 151
7.2 Beispiel . . . 152
7.3 Aufgaben . . . 167

Kapitel 8: Simulation . . . 169
8.1 Ein Maschinensimulator . . . 169
8.2 Code-Generierung im Speicher . . . 178
8.3 Beispiel . . . 187
8.4 Aufgaben . . . 189
8.5 Projekte . . . 190

Anhang A: „kleinC"-Compiler . . . 191
A.1 Syntaxanalyse . . . 191
A.2 Lexikalische Analyse . . . 201
A.3 Meldungen . . . 201
A.4 Symboltabelle . . . 203
A.5 Speichermanagement . . . 213
A.6 Code-Generierung . . . 215
A.7 Simulation . . . 222
A.8 Bessere Fehlermeldungen aus „yaccpar" . . . 235
A.9 Bessere Ablaufverfolgung von „yaccpar" . . . 239
A.10 Systematische Tests . . . 241

Literaturverzeichnis . . . 243

Sachverzeichnis . . . 245

Einführung

Bessere Benutzer-Schnittstellen, insbesondere für die vielen Mikro-Computer, die jetzt so populär werden, bedingen oft die Erkennung und Verarbeitung von recht aufwendigen Kommandosprachen. Spracherkennung (im grammatischen Sinn) hat also vielfältige Anwendungen, kann aber `auf die Schnelle´ fast unangenehm schwierig werden. Die Konstruktion eines Compilers illustriert eine wichtige Anwendung der Techniken zur Spracherkennung. Sie zeigt aber auch, wie man ein großes Programm entwirft und realisiert, unter Einsatz von Programmierwerkzeugen und durch schrittweise Entwicklung eines ersten Entwurfs. Einen Compiler verstehen zu lernen ist selbst schon interessant: man meistert dadurch ganz allgemein Programmiersysteme leichter und kann sie effizienter einsetzen.

Dieses Buch ist eine Fallstudie: es zeigt, wie man man einen Compiler entwikkelt, unter Verwendung von Programmgeneratoren wie *yacc* (für LALR(1) Parser) und *lex* (für lexikalische Analyse auf der Basis von regulären Ausdrücken). Dabei handelt es sich um zwei sehr vielseitige und dabei doch relativ leicht verwendbare Programmierwerkzeuge, die im Zusammenhang mit dem UNIX[1] System verfügbar sind.

Eine sehr einfache Teilmenge der Programmiersprache C, im Folgenden *kleinC* genannt, dient fortlaufend als Anschauungsobjekt zum Compilerbau. Die hier vorgestellte Implementierung von *kleinC* ist nicht als Ziel an sich gedacht; sie produziert deshalb auch nicht gerade optimalen Maschinen-Code. Ideen zur Verbesserung der Code-Generierung und für Spracherweiterungen finden sich in einigen Kapiteln als Aufgaben für den Leser.

Der Text vermeidet im Großen und Ganzen die Diskussion theoretischer Details, wie etwa eine ausführliche Darstellung des Grammatikbegriffs oder Erklärungen zum Innenleben und zur Funktionsweise der Generatoren. Entsprechende Hinweise auf die Literatur sind jedoch vorhanden. Wenigstens die einfacheren Aspekte zum Gebrauch der Generatoren werden ausführlich besprochen.

Als Konsequenz zeigen wir auf einer Ebene, wie man die Generatoren verwendet um eine einfache, leicht zu modifizierende Implementierung schnell und verläßlich vorzunehmen. Auf einer anderen Ebene erfährt der Leser Details über die Komponenten eines Compilers und über die üblichen Schnittstellen zwischen diesen Komponenten.

---

[1] UNIX ist ein eingetragenes Warenzeichen der Bell Laboratorien. Das UNIX Betriebssystem wird (für Europa) in Lizenz vergeben durch UNIX Europe limited, London.

Der Text ist daher sowohl als kurze Einführung gedacht, die einem detaillierten Studium von Algorithmen zum Compilerbau vorangehen sollte, als auch als Beschreibung, wie man speziell die erwähnten Generatoren erfolgreich einsetzt. Der Text soll weder alle Probleme beim Compilerbau umfassend behandeln, noch sollen alle Aspekte beim Gebrauch der Generatoren diskutiert werden. Nach Lektüre des vorliegenden Buchs sollte der Leser die Originalartikel über die Generatoren, [Joh78] und [Les78b], verstehen können; diese Artikel sind auch als Nachschlagewerk zu unserem Buch gedacht. Da ein Compiler ein relativ großes Programm ist, zeigt der Text, wie man ein solches Programm strukturiert und entwirft, unter Berücksichtigung von Problemen wie Fehlersuche, Erweiterung und Programmpflege.

Der Leser sollte einen Texteditor bedienen können, die üblichen Manipulationen mit Dateien ausführen können, und er muß natürlich in C programmieren können, denn in dieser Sprache sind alle Beispiele formuliert. Rudimentäre Kenntnis anderer Programmiersprachen, wie Pascal, Fortran oder Basic, wäre ebenfalls nützlich. Beginnend mit Kapitel 5 benötigen wir auch die Regeln zur Lebensdauer von Werten und zum Geltungsbereich von Namen bei Sprachen, die wie C der Algol-Familie entstammen. Bei der Implementierung werden dann auch einfache Datenstrukturen, wie etwa Stacks, ohne weitere Erklärungen verwendet. Kapitel 2 ist leichter zu verstehen, wenn der Leser eine gewisse Erfahrung mit Mustererkennung besitzt, etwa durch Umgang mit Thompson's *ed*-Editor.

Ein paar Bemerkungen zur Terminologie. Wir unterscheiden sorgfältig die *Deklaration* einer Variablen, Funktion, Marke oder eines anderen Programmobjekts von der *Definition* eines solchen Objekts. Im Deutschen werden beide Begriffe noch als *Vereinbarung* zusammengefaßt (A.d.Ü.). Eine Deklaration hält nur fest, daß ein Programmobjekt existiert und gewisse Eigenschaften besitzt, wie den Typ einer Variablen, die Typen von Parametern und dem Resultat einer Funktion, usw. Eine Definition sorgt auch noch für die Existenz eines Programmobjekts, wobei eine Variable zum Beispiel einen Initialisierungswert erhalten kann, für eine Funktion eine Folge von Anweisungen angegeben wird, eine Marke mit einem bestimmten Punkt im Programm verknüpft wird, usw.

Bei der Beschreibung von Grammatiken sind wir geringfügig vom allgemeinen Sprachgebrauch abgewichen. Wir wollen die Begriffe *Produktion*, *rechte Seite* und *linke Seite* sowie (im Deutschen) *terminale* und *nicht-terminale* Symbole vermeiden. An Stelle dieser Begriffe verstehen wir unter einer *Regel* alle Produktionen (mit allen Alternativen auf der rechten Seite) für jeweils ein nicht-terminales Symbol. Eine *Formulierung* ist dann genau eine der möglichen rechten Seiten. Wir sprechen außerdem von *Eingabesymbolen* und von *Grammatikbegriffen*.

Das *kleinC*-Beispiel kann vom zweiten Autor, zum Beispiel über ARPANET, BITNET, CSNET oder USENET, oder auch von den Übersetzern bezogen werden.

Dem Gutachter danken wir für eine sehr konstruktive Kritik. Wir sind außerdem dankbar, daß wir die Computer im Department of Computer Science der University of Illinois in Urbana-Champaign und bei der Sektion Informatik der Universität Ulm für Entwicklung, Textgestaltung und Lichtsatz benutzen durften. Die Anlage in Urbana wird teilweise von der National Science Foundation (MCS 81-05896) unterstützt.

Die deutsche Ausgabe entstand unter Verwendung von Computern der Sektion Informatik der Universität Ulm und der ursprünglichen Quellen (A.d.Ü.).

A.T.Schreiner
H.G.Friedman
Oktober, 1984
Urbana, Illinois

A.T.Schreiner
E.Janich
Februar, 1985
Ulm

Kapitel 1: Sprachdefinition

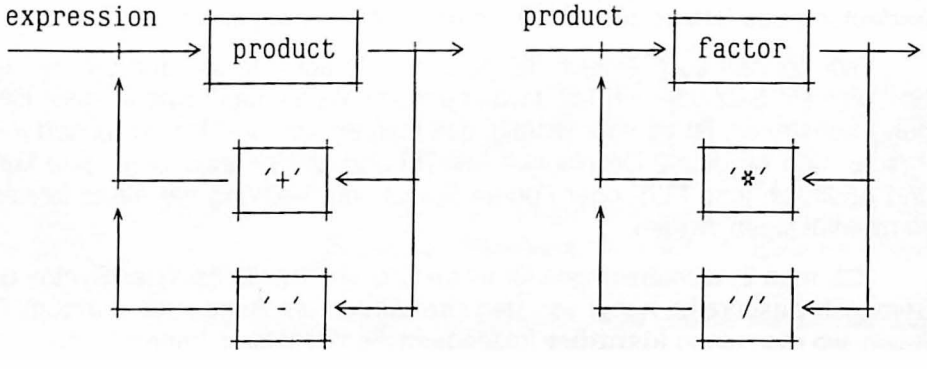
Ein Compiler akzeptiert ein *Quellprogramm* - ein Programm, das in einer *Quellsprache* formuliert ist - und konstruiert daraus ein *Objektprogramm*, das in einer *Objektsprache* wie Assembler oder Maschinencode geschrieben ist. Anders ausgedrückt: ein Compiler muß als Eingabe ein Quellprogramm erkennen, auf Konsistenz in bezug auf Grammatik und Bedeutung überprüfen, und dann ein äquivalentes Objektprogramm konstruieren.

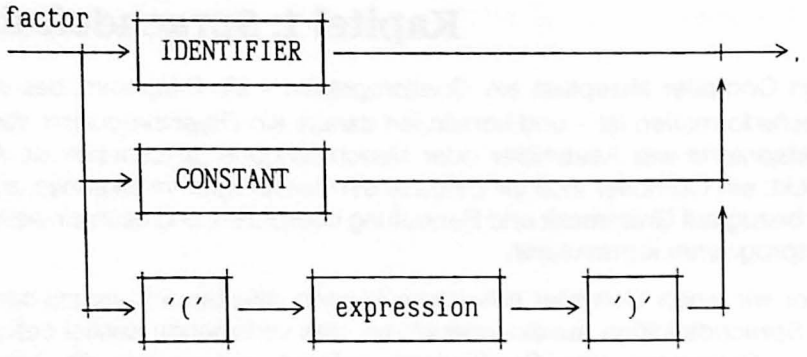
Bevor wir einen Compiler entwickeln können, müssen wir uns mit dem Problem der Sprachdefinition auseinandersetzen. Das vorliegende Kapitel beschreibt, wie man die Grammatik einer Quellsprache definiert, und wie man die Grammatik selbst überprüft. Im nächsten Kapitel untersuchen wir, wie man die einzelnen 'Worte' der Quellsprache erkennen kann. Im Kapitel 3 kombinieren wir dann diese zwei Aspekte eines Compilers, so daß er ein korrektes Quellprogramm erkennen kann und im Kapitel 4 schließlich erweitern wir den Compiler, damit er sich auch dann 'vernünftig' verhält, wenn das Quellprogramm Fehler enthält. Die anschließenden Kapitel zeigen, wie man die Information speichert, die man bei der Übersetzung braucht, und wie man diese Information verwendet um schließlich ein Objektprogramm zu generieren.

1.1 Zweck

Eine Sprachdefinition hat zwei Aspekte: *Syntax* und *Semantik*. Die Syntax behandelt die formalen Probleme, nämlich ob eine Folge von Worten (oder Zeichen) einen *Satz* der Sprache bildet. Was der Satz bedeutet - und manchmal auch, ob der Satz in dieser Hinsicht akzeptabel ist oder nicht - wird von der Semantik der Sprache festgelegt.

Beide Teile der Sprachdefinition können mit Formalismen beschrieben werden. Die Syntax erklärt man normalerweise mit Hilfe einer Folge von Modellen, die die einzelnen Teile eines Satzes beschreiben. *Syntaxgraphen*, die Wirth erstmals bei der Definition von Pascal verwendet hat, werden ähnlich wie Flußdiagramme gezeichnet:





Backus Naur Form (BNF) ist eine Sprache, mit der die Syntax einer Sprache (auch von BNF selbst) angegeben werden kann:

```
expression
        : expression '+' product
        | expression '-' product
        | product

product
        : product '*' factor
        | product '/' factor
        | factor

factor
        : IDENTIFIER
        | CONSTANT
        | '(' expression ')'
```

Wir werden BNF später noch genauer betrachten.

Die Semantik einer Sprache zu beschreiben ist wesentlich schwieriger. Wird ein Formalismus verwendet, simuliert er im wesentlichen die Ausführung eines Satzes auf einem mehr oder weniger wohldefinierten theoretischen Maschinenmodell. Die resultierenden Zustände der Maschine - ob legitim oder nicht - definieren die Bedeutung des Satzes oder verbieten den Satz als bedeutungslos.

Wir können auch einfach die deutsche Sprache verwenden um zu beschreiben, was ein Satz oder ein Teil davon bedeutet. Wenn man keine formale Beschreibung konstruiert, ist es sehr wichtig, daß man einfach und klar formuliert - Wirth's Pascal- und Modula-2-Definitionen [Jen75] und [Wir82] sind sehr gute Beispiele, und praktisch jede PL/I- oder Fortran-Sprachbeschreibung hat einen fatalen Hang zu überflüssigen Worten.

Ob man Einschränkungen für einen Satz eher mit Mitteln der Syntax oder der Semantik beschreibt, hängt von den Umständen ab. Betrachten wir zum Beispiel Basic, wo man einen **identifier** folgendermaßen definieren kann:

```
id
        : LETTER DIGIT
        | LETTER

identifier
        : id '$'
        | id
```

Wird der Operator + für Addition bei numerischen Werten *und* zur Verkettung von Zeichenketten verwendet, muß die Semantikbeschreibung festlegen, daß + nicht dazu verwendet werden darf, eine Zeichenkette zu einem numerischen Wert zu addieren oder umgekehrt.

Wir können die Syntax aber auch anders definieren:

```
real_id
        : LETTER DIGIT
        | LETTER

string_id
        : real_id '$'
```

So kann man rein auf der Basis der Syntax in der ganzen Sprachdefinition zwischen Operationen mit Zeichenketten und mit numerischen Werten unterscheiden.

Ein **identifier** in Basic ist sicher ein Grenzfall zwischen dem, was man syntaktisch beschreiben kann und soll, und dem, was man lieber in der Semantikdefinition festlegt. Im allgemeinen sollte man bei einer Sprachdefinition nicht mit deutschen Sätzen beschreiben, was man (vernünftigerweise) formaler ausdrücken kann.

1.2 Formale Aspekte

BNF ist ein Formalismus zur Beschreibung der Syntax einer Sprache. BNF wurde zuerst von Peter Naur im Algol 60 Report [Nau63] verwendet und ist seither für viele Sprachdefinitionen benutzt worden. Der Formalismus ist auch erweitert und überladen worden, vielleicht bereits bis zu einem Punkt, an dem die resultierende Sprachbeschreibung nicht mehr leicht zu verstehen ist [Wij75].

Eine *Grammatik*, die BNF Definition einer Sprache, besteht aus einer Folge von *Regeln*. Eine Regel besteht aus einer *linken Seite* und einer *rechten Seite*, die durch einen Doppelpunkt getrennt sind. Die linke Seite besteht aus einem einzigen, eindeutigen *Grammatikbegriff* (*non-terminal symbol*).[1] Die rechte Seite besteht aus einer Folge von einer oder mehreren *Formulierungen*, die voneinander durch einen Strich | getrennt sind. Eine Formulierung in einer Regel kann leer sein; sonst besteht eine Formulierung aus einer Folge von Grammatikbegriffen oder *Eingabesymbolen*

[1] Wir beschreiben hier nur kontext-freie Sprachen, und dies auch nur sehr informell. Als definitive Quelle kann zum Beispiel [Aho77] dienen.

(*terminal symbols*). Grammatikbegriffe und Eingabesymbole bezeichnen wir zusammen auch als *Symbole*. Da eine Regel nur eine Formulierung benötigt, und da diese Formulierung leer sein kann, kann die rechte Seite einer Regel leer sein. Ein Beispiel einer Grammatik wurde schon im Abschnitt 1.1 gezeigt:

```
expression
        : expression '+' product
        | expression '-' product
        | product

product
        : product '*' factor
        | product '/' factor
        | factor

factor
        : IDENTIFIER
        | CONSTANT
        | '(' expression ')'
```

Eine Grammatik beschreibt eine Sprache dadurch, daß sie erklärt, welche Sätze gebildet werden können. Der Grammatikbegriff auf der linken Seite der ersten Regel wird als *Startsymbol* bezeichnet. In diesem Beispiel ist **expression** das Startsymbol. Die erste Regel enthält alle Formulierungen für das Startsymbol. Jede Formulierung kann neue Grammatikbegriffe einführen, wie hier etwa **product**.

Für jeden Grammatikbegriff muß eine Regel existieren, und jede Formulierung aus dieser Regel kann an Stelle des Grammatikbegriffs eingefügt werden. Diese Ersetzung geschieht solange, bis eine Folge von Eingabesymbolen, ein *Satz*, aus dem Startsymbol der Grammatik *produziert* wurde. Eingabesymbole werden durch die Grammatik nicht näher erklärt; sie bilden das Alphabet, in dem die Sätze der Sprache geschrieben werden, die die Grammatik beschreibt.

Wenn die Regeln einer Grammatik Grammatikbegriffe rekursiv verwenden, können unendlich viele - und unendlich lange - Sätze gebildet werden. In unserem Beispiel ist zum Beispiel folgender Satz möglich

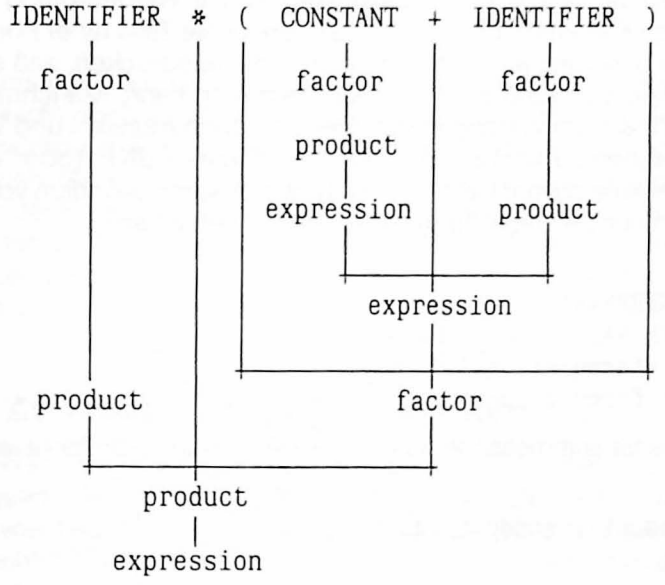
```
IDENTIFIER
```

wenn man die Formulierung **product** für **expression**, **factor** für **product**, und **IDENTIFIER** für **factor** wählt. Ein weiteres Beispiel ist

```
IDENTIFIER * ( CONSTANT + IDENTIFIER )
```

Hier benötigen wir eine Reihe von Zwischenstufen, bevor wirklich feststeht, daß tatsächlich ein Satz im Sinne der Grammatik vorliegt. Üblicherweise arrangiert man diese Formulierungen als *Syntaxbaum*. Die Wurzel dieses Baumes ist das Startsymbol der Grammatik. Die Blätter sind der Reihe nach die Eingabesymbole eines Satzes. Jeder Knoten, mit Ausnahme der Blätter, entspricht einem Grammatikbegriff, und

Kanten führen von einem Grammatikbegriff (Knoten) zu anderen Knoten, die der Reihe nach den Symbolen einer Formulierung für diesen Grammatikbegriff entsprechen.



BNF ist selbst eine Sprache und kann in BNF definiert werden:

```
grammar
        : grammar rule
        | rule

rule
        : rule '|' formulation
        | NONTERMINAL ':' formulation
        | NONTERMINAL ':'

formulation
        : formulation symbol
        | symbol

symbol
        : NONTERMINAL
        | TERMINAL
```

Diese Definition ist mehr eingeschränkt als die frühere, informelle Beschreibung: hier muß eine leere Formulierung die erste in einer Regel sein. (Warum?!)

Wie das Beispiel zeigt, spielt Rekursion in BNF eine sehr große Rolle. Man schreibt Regeln meist *links-rekursiv* - der zu definierende Grammatikbegriff er-

scheint wieder am Anfang einer seiner eigenen Formulierungen, wodurch eine beliebig lange Folge gleicher Phrasen produziert werden kann. Diese Technik macht an sich einfache Situationen leicht undurchschaubar. Insbesondere in Sprachhandbüchern wird BNF deshalb oft mit speziellen Konstruktionen zur Wiederholung erweitert, wie etwa mit eckigen Klammern [und], die dann optionale Teile einer Formulierung umgeben, und mit geschweiften Klammern { und }, die ausdrücken, daß ein Teil einer Formulierung beliebig oft (und auch gar nicht) auftreten kann. Manchmal verwendet man auch Klammern um Vorrang einzuführen und dann Auswahl und Verkettung von Symbolen trennen zu können. *Extended BNF*, kurz EBNF, (oder wenigstens unsere private Variante davon) kann man leicht aus unserer Definition von BNF entwickeln. Wir müssen nur die Regel für **symbol** wie folgt ersetzen:

```
symbol
        : NONTERMINAL
        | TERMINAL
        | '{' formulation '}'
        | '[' formulation ']'
```

Unsere Grammatik für arithmetische Ausdrücke kann dann vereinfacht werden:

```
expression
        : product { addop product }

addop
        : '+'
        | '-'

product
        : factor { mulop factor }

mulop
        : '*'
        | '/'
```

factor bleibt wie vorher.

Auch EBNF kann sich selbst beschreiben:

```
grammar
        : rule { rule }

rule
        : NONTERMINAL ':' [ formulation ] { '|' formulation }

formulation
        : symbol { symbol }
```

```
symbol
        : NONTERMINAL
        | TERMINAL
        | '{' formulation '}'
        | '[' formulation ']'
```

Unsere Diskussion war zwar ganz informell, aber ein Wort zur Repräsentierung ist vielleicht doch angebracht: *Grammatikbegriffe* gleichen den Namen in einer Programmiersprache, sie bestehen also aus Buchstaben und vielleicht Ziffern und _-Zeichen. Sie müssen mit einem Buchstaben beginnen. *Eingabesymbole* benennt man in der Regel mit Großbuchstaben. Handelt es sich um einzelne Sonderzeichen, kann man sie auch direkt in einfachen Anführungsstrichen angeben. Zwischenraum (also Leerzeichen und Tabulatorzeichen) ist normalerweise bei der Definition einer Grammatik nicht signifikant und dient nur zur Trennung von benachbarten Symbolen.

1.3 Zweckmäßige Grammatiken

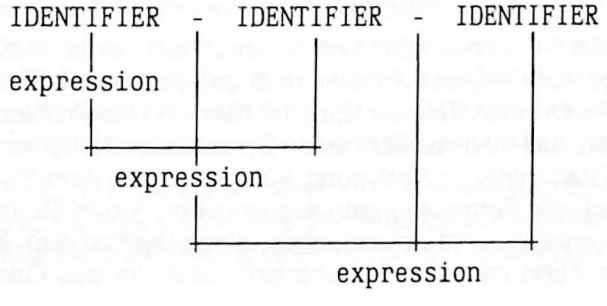
Wenn wir eine Sprache als Menge ihrer Sätze definieren, das heißt, als Menge von bestimmten Folgen von Eingabesymbolen, dann kann man normalerweise eine Reihe von Grammatiken konstruieren, die die Sprache beschreiben. Eine Sprache muß nicht endlich sein, eine Grammatik muß nach Definition aber endlich sein. Diese Einschränkung allein genügt aber nicht. Betrachten wir einmal folgende zwei Grammatiken, die beide sehr primitive arithmetische Ausdrücke beschreiben:

```
expression
        : expression '-' IDENTIFIER
        | IDENTIFIER
```

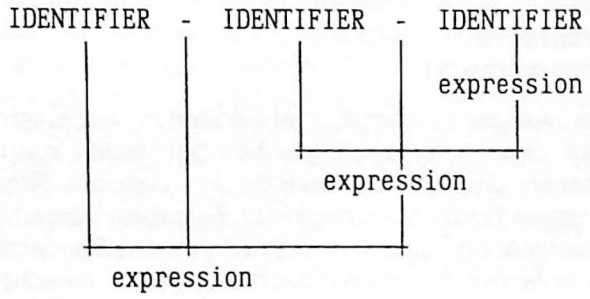
und

```
expression
        : IDENTIFIER '-' expression
        | IDENTIFIER
```

Bei beiden Grammatiken besteht die Sprache aus allen alternierenden Folgen von **IDENTIFIER** und −. Es gibt jedoch einen wichtigen Unterschied, den wir erkennen, wenn wir die Syntaxbäume für einen bestimmten Ausdruck vergleichen:



Die links-rekursive Grammatik sammelt Eingabesymbole von links her. Die rechts-rekursive Grammatik führt hier zu einem spiegelverkehrten Syntaxbaum:



Speziell im Kontext arithmetischer Ausdrücke *interpretiert* man den Syntaxbaum, damit er zusätzlich noch Vorrang und Assoziativität von Operatoren festlegt, also die Reihenfolge, in der die Teile eines arithmetischen Ausdrucks zur Bewertung zusammengefaßt werden. Das vorstehende Beispiel führt nur einen Operator ein, es gibt daher kein Vorrangsproblem. Je nach Grammatik würden wir aber verschieden implizit klammern: die links-rekursive Grammatik impliziert, daß der Operator – links-assoziativ ist, das heißt, daß implizit Klammern von links her eingefügt werden, und die rechts-rekursive Grammatik verlangt Klammern von rechts her. Die meisten Operatoren sind links-assoziativ und sollten daher links-rekursiv definiert werden.

Dies ist aber leider nicht das einzige Problem. Betrachten wir folgenden Vorschlag für eine Grammatik, die immer noch die gleiche Sprache mit arithmetischen Ausdrücken beschreiben soll:

```
expression
        : expression '-' expression
        | IDENTIFIER
```

Mit dieser Regel können wir *jeden* der beiden Syntaxbäume entwickeln. Diese Grammatik nennt man daher *mehrdeutig*: es gibt einen Satz in der Sprache, für den man zwei verschiedene Syntaxbäume entwickeln kann. Mehrdeutigkeit ist oft eine Eigenschaft der Grammatik und normalerweise nicht der Sprache. Wie das Beispiel zeigt, gibt es hier auch eindeutige Grammatiken für die Sprache. Wir können den letzten Vorschlag dadurch eindeutig machen, daß wir verlangen, daß der Syntaxbaum von links her zur Wurzel hin aufgebaut werden muß – eine solche Bedingung nennen wir eine *Vorschrift gegen Mehrdeutigkeit* (*disambiguating rule*).

Als erste Aufgabe muß ein Compiler die Sprache erkennen, die er übersetzen soll, das heißt, für jede Folge von Eingabesymbolen muß der Compiler effizient entscheiden können, ob es sich um einen Satz der Sprache handelt. Das Problem kann dadurch angegangen werden, daß man versucht einen Syntaxbaum zu konstruieren. Haben wir eine eindeutige Grammatik zur Verfügung, können wir mit dem Startsymbol beginnen, der Reihe nach alle Formulierungen ausprobieren, jeden Grammatikbegriff entsprechend weiter ersetzen und so versuchen, einen Syntaxbaum für eine Eingabefolge zu entwickeln. Führt man dies konsequent durch, ist das Ganze ein

backtracking Verfahren und daher von vornherein ineffizient. Wenn der Compiler auch noch eine Ausgabe liefert, während er versucht den Syntaxbaum aufzubauen, ist *backtracking* nicht einmal ohne Weiteres möglich, denn man müßte ja auch einen Teil der Ausgabe wieder rückgängig machen!

Eine zweckmäßige Grammatik zur Spracherkennung muß nicht nur eindeutig sein, sie muß deterministisch sein: der bisher unbenutzte Rest der Eingabe, das heißt, der Rest der Folge von Eingabesymbolen, die insgesamt ein Satz sein soll, zusammen mit dem teilweise konstruierten Syntaxbaum, muß uns in die Lage versetzen, eindeutig zu entscheiden, welche Regel und Formulierung wir benutzen müssen um den Syntaxbaum zu vervollständigen falls er existiert, oder zu erkennen, daß kein Baum entwickelt werden kann, der auch noch das nächste Eingabesymbol beinhalten kann.

Verschiedene Ansichten zur Zweckmäßigkeit einer Grammatik existieren. Sie hängen stark vom Einfallsreichtum des Compiler-Bauers und der Vielseitigkeit seiner Werkzeuge ab. Eine einfache hinreichende Bedingung besteht zum Beispiel darin, daß jede Formulierung mit einem eindeutigen Eingabesymbol beginnen soll. In diesem Fall bestimmt das nächste, einzelne Eingabesymbol, was jeweils zu tun ist.

Die Grammatiken moderner Programmiersprachen folgen im allgemeinen einem weniger eingeschränkten Muster: sie haben normalerweise die sogenannte LL(1) oder LR(1) Eigenschaft oder eine Variante davon. Im ersten Fall, also bei Vorliegen der LL(1) Eigenschaft, kann man einen Syntaxbaum vom Startsymbol zu den Eingabesymbolen hin (*top-down*) ohne *backtracking* entwickeln; im zweiten Fall, bei LR(1), kann man einen Syntaxbaum in der umgekehrten Richtung (*bottom-up*) ohne *backtracking* mit Hilfe von Tabellen erstellen, für die es Konstruktionsprogramme gibt. In jedem dieser Fälle müssen wir immer nur das jeweils *nächste* Symbol in der Eingabe kennen, das heißt wir benötigen *ein Symbol look-ahead*.

LL(1) ist schlicht folgende Eigenschaft: immer wenn es fraglich ist, welche Regel oder Formulierung verwendet werden sollte, muß das nächste Eingabesymbol uns eine eindeutige Entscheidung ermöglichen. Die Frage taucht auf, wenn wir eine aus mehreren Formulierungen in einer Regel auswählen müssen. Ist keine Formulierung leer, beginnt jede entweder mit einem Eingabesymbol oder mit einem Grammatikbegriff. Das nächste Symbol in der Eingabe kann eines dieser Eingabesymbole sein. Alternativ dazu ergeben sich weitere Eingabesymbole am Anfang von Formulierungen, wenn wir rekursiv alle Formulierungen für alle Grammatikbegriffe am Anfang der Formulierungen der ursprünglichen Regel untersuchen. Das nächste Symbol in der Eingabe *muß genau eines* dieser Eingabesymbole sein, die folglich alle voneinander verschieden sein müssen. Gibt es eine leere Formulierung im obigen Fall, müssen wir auch noch alle einleitenden Eingabesymbole berücksichtigen für alle Formulierungen von Grammatikbegriffen, die in diesem Fall *folgen* können.

Die LL(1) Eigenschaft bei einer Grammatik nachzuweisen ist in der Praxis nicht schwierig, da man sie mit Hilfe von Relationen formulieren kann wie etwa *ein Eingabesymbol steht am Anfang einer Formulierung für einen Grammatikbegriff* oder *zwei*

Symbole folgen einander in einer Formulierung. Derartige Relationen kann man als Boole'sche Matrizen darstellen [Gri71], und kompliziertere Relationen können dann zusammengesetzt und berechnet werden. Die LL(1) Eigenschaft nutzbringend zu verwenden ist noch einfacher: hat eine Grammatik die LL(1) Eigenschaft, kann man mit rekursiv aufgerufenen Funktionen direkt aus der Grammatik einen Erkennungsalgorithmus für die zugehörige Sprache implementieren (siehe zum Beispiel [Wir77]).

Auch wenn eine Grammatik absolut nicht die LL(1) Eigenschaft hat, kann man mit semantischen Tricks die Erkennung der Sprache oft doch noch deterministisch gestalten. Betrachten wir dazu einen vereinfachten Auszug aus Pascal:

```
statement
        : IDENTIFIER ':=' expression
        | IDENTIFIER
```

Die erste Formulierung beschreibt eine Zuweisung, die zweite einen Prozeduraufruf. Sind beide Versionen von **IDENTIFIER** wirklich gleich, kann man mit einem Namen als nächstem Symbol in der Eingabe nicht entscheiden, welche Formulierung verwendet werden soll. Das Problem umgeht man normalerweise dadurch, daß man sich klar macht, daß nur ein *procedure* **IDENTIFIER** am Anfang eines Prozeduraufrufs stehen kann. Ein derartiger **IDENTIFIER** kann aber eine Zuweisung nicht einleiten. Da man in Pascal Namen vereinbaren muß, bevor man sie verwenden kann, lösen wir so unser Syntaxproblem mit einem semantischen Trick.

1.4 Überprüfung einer Grammatik

Ein Compiler muß entscheiden, ob eine Folge von Eingabesymbolen ein Satz in einer Sprache ist. Eine Grammatik beschreibt eine Sprache. Eine offensichtliche Frage ist, ob wir eine Grammatik mehr oder weniger direkt zur Erkennung einer Sprache heranziehen können.

Zuerst aber sollten wir die Grammatik überprüfen: für jeden Grammatikbegriff muß eine Regel existieren, alle Grammatikbegriffe müssen vom Startsymbol her erreichbar sein und die Grammatik sollte eine Eigenschaft wie LL(1) besitzen, damit sie sich zur Spracherkennung eignet und eindeutig ist.

Johnson's *yacc*, ein leistungsfähiges Dienstprogramm im UNIX System [Joh78], kann benutzt werden um eine Grammatik derart zu überprüfen. Unter dem hier betrachteten Gesichtspunkt leistet *yacc* folgendes: *yacc* akzeptiert eine in BNF angegebene Grammatik und identifiziert alle Probleme, die die Grammatik zur Spracherkennung untauglich machen. Als Beispiel schreiben wir folgende Eingabedatei *grammar*:

```
/*
 *      erstes Beispiel einer yacc-Grammatik
 */

%token  IDENTIFIER

%%

expression
        : expression '-' IDENTIFIER
        | IDENTIFIER
```

und überprüfen sie mit dem Kommando

```
yacc grammar
```

Nichts geschieht, und das soll auch so sein - nach Ansicht von *yacc* ist die Grammatik akzeptabel. Tatsächlich besitzen wir im Anschluß an das Kommando eine neue Datei *y.tab.c*, die im Kapitel 3 weiter diskutiert wird.

Wie das Beispiel zeigt, verwendet man in der Eingabe für *yacc* die Repräsentierung von BNF, die wir früher eingeführt haben. Da Eingabesymbole und Grammatikbegriffe gleich aussehen, muß man für *yacc* die Eingabesymbole mit **%token**-Anweisungen vor der eigentlichen Grammatik vereinbaren. Eine Zeile mit **%%** muß die zwei Teile der Eingabedatei trennen.[1] Zwischenraum und Kommentare im Stil von C werden ignoriert.

Aus technischer Sicht wollen wir nun sehen, wie *yacc* eine Grammatik überprüft. (Wie die Berechnung tatsächlich verläuft wird von Horning im Kapitel 2.C. in [Bau76] oder in [Aho74] beschrieben.) *yacc* verfügt über eine Einrichtung zur Fehlersuche in einer Grammatik; wenn wir das Kommando

```
yacc -v grammar
```

ausführen, erhalten wir folgende Datei *y.output*:

```
state 0
        $accept : _expression $end

        IDENTIFIER  shift 2
        .  error

        expression  goto 1
```

[1] Wir führen die relevanten Aspekte der Eingabesprache für *yacc* im währenden ein. Zum Nachschlagen sollte der Leser [Joh78] verwenden.

```
state 1
        $accept :  expression_$end
        expression :  expression_- IDENTIFIER

        $end  accept
        -  shift 3
        .  error

state 2
        expression :  IDENTIFIER_    (2)

        .  reduce 2

state 3
        expression :  expression -_IDENTIFIER

        IDENTIFIER  shift 4
        .  error

state 4
        expression :  expression - IDENTIFIER_    (1)

        .  reduce 1

4/127 terminals, 1/175 nonterminals
3/350 grammar rules, 5/550 states
0 shift/reduce, 0 reduce/reduce conflicts reported
3/225 working sets used
memory: states,etc. 26/4500, parser 0/3500
3/400 distinct lookahead sets
0 extra closures
3 shift entries, 1 exceptions
1 goto entries
0 entries saved by goto default
Optimizer space used: input 9/4500, output 4/3500
4 table entries, 0 zero
maximum spread: 257, maximum offset: 257
```

Wir zeigen hier zwar die ganze Datei *y.output*, aber nur die ersten paar Zeilen bei jedem **state** sind im Augenblick wichtig. *y.output* wird im Abschnitt 3.1 noch mehr im Detail betrachtet.

yacc fügt die Regel

```
$accept : expression $end
```

zur Grammatik hinzu und markiert eine Position (in der Regel durch einen Unterstrich _ kenntlich gemacht) unmittelbar vor dem Startsymbol, das heißt, hier vor **expression**. Das Ganze nennt man **state 0**. *yacc* versucht nun, die Positionsmarkierung über das Startsymbol hinweg zu verlagern.

Immer wenn eine Position vor einem Grammatikbegriff markiert ist, wird gleichzeitig und parallel dazu auch die Position vor jeder Formulierung des Grammatikbegriffs markiert. Eine Menge von Formulierungen mit markierten Positionen, die so entsteht, bezeichnet man als *Zustand* (*state*). Eine Formulierung mit einer markierten Position nennt man eine *Konfiguration*.

Einen neuen Zustand berechnet man nun folgendermaßen: im alten Zustand bewegt man der Reihe nach in jeder Konfiguration die Positionsmarkierung über das nächste Symbol hinweg. Im alten Zustand macht man dies für alle Konfigurationen mit gleichem nächsten Symbol gleichzeitig. Für jedes mögliche Symbol ergibt sich so eine neue Menge von Konfigurationen, die man als neuen Zustand bezeichnet.

Wie vorher beschrieben wurde, muß man, wenn in einer Konfiguration eine Position vor einem Grammatikbegriff markiert wird, alle Formulierungen für diesen Grammatikbegriff zum Zustand hinzunehmen, wobei jeweils der Anfang dieser neuen Formulierungen markiert wird.

Das Verfahren muß abbrechen, da nach Definition nur eine endliche Anzahl von Regeln und Grammatikbegriffen, von Formulierungen und von Eingabesymbolen in einer Grammatik vorliegen kann, und da die Formulierungen nicht unendlich viele Symbole enthalten dürfen. Es entsteht also eine endliche Zahl von Zuständen.

Im Beispiel wird **state 0** implizit von *yacc* konstruiert. Da die Position vor dem Grammatikbegriff **expression** markiert ist, müssen wir alle Formulierungen von **expression** mit einer Markierung am Anfang jeder solchen Formulierung hinzunehmen:

```
expression : _expression - IDENTIFIER
expression : _IDENTIFIER
```

Dies wird in *y.output* nicht gezeigt, da es für den von *yacc* verwendeten Algorithmus offensichtlich dazugehört.

In **state 0** sind nun Positionen vor **expression** und **IDENTIFIER** markiert. Bewegt man die Markierung über **expression** in allen möglichen Konfigurationen hinweg, erhält man **state 1**. Überquert man **IDENTIFIER**, gelangt man in **state 2**. In keinem der beiden Zustände geht die Markierung jetzt einem Grammatikbegriff voraus, also muß man keine neuen Konfigurationen berücksichtigen. **state 2** enthält nur eine *vollständige Konfiguration*, bei der die Positionsmarkierung das Ende einer Formulierung erreicht hat.

Im **state 1** können wir mit der Markierung das fiktive **$end**-Symbol überqueren, das *yacc* eingeführt hat, oder das Eingabesymbol – in der anderen Konfiguration. Die erste Operation wird im wesentlichen ignoriert; die zweite produziert **state 3**.

Auch **state 3** kann nicht mehr erweitert werden und wir können nur noch **IDENTIFIER** überqueren, womit wir **state 4** erreichen. Dies ist der letzte mögliche Zustand, da er nur noch eine vollständige Konfiguration enthält.

Dieser Algorithmus simuliert natürlich die Verwendung aller Teile der Grammatik zur Erkennung einer Eingabe. Die Überquerung eines Eingabesymbols in einer Formulierung, eine sogenannte *shift*-Operation, entspricht dem Akzeptieren dieses Symbols an einer bestimmten Position in einer Eingabefolge. Die Überquerung eines Grammatikbegriffs bedingt, daß eine Formulierung für den Grammatikbegriff an einer anderen Stelle in der Simulation vollständig überquert wurde. (Wir verfolgen dies im Kapitel 3 noch näher.)

Die Kommentare, die den Konfigurationen bei jedem **state** in *y.output* folgen, skizzieren, welche Operationen im Zusammenhang mit jedem möglichen Eingabesymbol durchgeführt werden würden. Ein Punkt steht dabei für *jedes andere Symbol*. **reduce** gibt an, daß eine vollständige Konfiguration, also eine bestimmte Formulierung, dazu verwendet wird, eine Anzahl von akzeptierten Symbolen zusammenzufassen und durch den entsprechenden Grammatikbegriff zu ersetzen. Bei einer **reduce**-Operation steht jeweils die Nummer der Formulierung, die verwendet wird. Bei **shift** ist der Zustand angegeben, in den man gelangt, wenn das entsprechende Symbol akzeptiert wird.

1.5 Fallstricke

Es sieht zwar so aus, als ob bei dieser Analyse nichts schiefgehen kann, aber dies ist nicht der Fall. Vereinbart man ein Eingabesymbol nicht, oder hat man für einen Grammatikbegriff keine Regel definiert, oder gibt es Grammatikbegriffe und Regeln, die vom Startsymbol her nicht erreicht werden können, so gibt *yacc* sofort entsprechende Fehlermeldungen aus.

Probleme in der Grammatik selbst sind subtiler und müssen individuell untersucht werden. Betrachten wir folgende kleine Grammatik, die die übliche **if**-Anweisung definiert:

```
/*
 *      `else' Problem
 */

%token  IF...THEN
%token  ELSE

%%

statement
        : ';'           /* leere Anweisung */
        | IF...THEN statement
        | IF...THEN statement ELSE statement
```

yacc beschwert sich über einen *shift/reduce*-Konflikt und in *y.output* finden wir folgendes:

```
4: shift/reduce conflict (shift 5, red'n 2) on ELSE
state 4
        statement :  IF...THEN statement_    (2)
        statement :  IF...THEN statement_ELSE statement

        ELSE  shift 5
        .  reduce 2

state 5
        statement :  IF...THEN statement ELSE_statement

        IF...THEN  shift 3
        ;  shift 2
        .  error

        statement  goto 6
```

Das Problem tritt häufig auf. Es entspricht der Frage, zu welchem **if** im folgenden Programmfragment **else** eigentlich gehört:

```
if (condition)
        if (condition)
                ;                /* leere Anweisung */
        else
                ...
```

Aus der Sicht von *yacc* können wir in **state 4 ELSE** noch akzeptieren (eine **shift**-Operation) und in der zweiten Konfiguration fortschreiten, oder wir könnten die erste, vollständige Konfiguration verwenden und den Grammatikbegriff **statement** einsetzen (eine **reduce**-Operation). Durch eine **shift**-Operation verlängern wir die innerste **if**-Anweisung, das heißt wir ordnen **else** dem innersten **if** zu, und das wird in Sprachen wie C und Pascal ja auch verlangt.

Technisch liegt bei **state 4** ein *shift/reduce*-Konflikt vor, also ein Fehler in der Grammatik; die Grammatik ist mehrdeutig und damit zur Spracherkennung nicht geeignet. *yacc* jedoch läßt derartige Konflikte absichtlich zu und sorgt (wie aus den Kommentaren hervorgeht) dafür, daß die längstmögliche Eingabefolge zusammengefaßt wird, das heißt, es passiert genau das, was man im Normalfall möchte.

Das nächste Problem ist ernster. Wir betrachten es im Rahmen eines Auszugs aus einem Basic Dialekt, bei dem das Resultat eines Vergleichs an eine Variable zugewiesen oder auch für eine **if**-Anweisung verwendet werden kann.

```
/*
 *      Mehrfachklauseln
 */

%token  IDENTIFIER, IF, THEN, SUM

%%

statement
        : IDENTIFIER '=' expression
        | IF condition THEN statement

condition
        : expression
        | SUM '<' SUM

expression
        : SUM '<' SUM
        | SUM
```

yacc bemängelt einen *reduce/reduce*-Konflikt und Nachforschungen in *y.output* ergeben folgendes:

```
state 14
        condition :  SUM < SUM_    (4)
        expression :  SUM < SUM_    (5)

        .  reduce 4
```

state 14 enthält mehr als eine vollständige Konfiguration. In diesem Fall würde *yacc* die Formulierung verwenden, die zuerst in der Grammatik angegeben wurde, aber das ist riskant, falls die Grammatik später modifiziert oder gar anders angeordnet wird. Während wir *shift/reduce*-Konflikte wie bei **else** normalerweise tolerieren, versuchen wir immer, eine Grammatik so umzuschreiben, daß wir Kombinationen von Formulierungen vermeiden, die *reduce/reduce*-Konflikte verursachen. Im vorliegenden Fall ist das sehr leicht möglich - zugegeben, das Beispiel ist zu einfach um realistisch zu sein:

```
/*
 *      Mehrfachklauseln, ohne Konflikte
 */

%token  IDENTIFIER, IF, THEN, SUM
```

```
%%

statement
        : IDENTIFIER '=' expression
        | IF expression THEN statement

expression
        : SUM '<' SUM
        | SUM
```

yacc ist sehr nachsichtig in seiner eigenen Eingabesyntax. Zur Platzersparnis in diesem Buch haben wir sogar das Semikolon weggelassen, das jeder Regel folgen kann. Fügt man versehentlich bei einer Regel eine leere Formulierung hinzu, kann dies zu einer überraschend hohen Zahl von Konflikten führen!

Ein weiteres Mehrdeutigkeitsproblem erweist sich tatsächlich als Vorteil. Betrachten wir folgende Grammatik für arithmetische Ausdrücke:

```
expression
        : expression '+' expression
        | expression '-' expression
        | expression '*' expression
        | expression '/' expression
        | IDENTIFIER
```

Die Definition ist zwar kurz und enthält keine 'überflüssigen' Grammatikbegriffe, aber sie gibt keinerlei Hinweise auf Vorrang und Assoziativität der Operatoren. *yacc* erlaubt eine derartige Definition, wenn man Vorrang und Assoziativität explizit im ersten Teil der Definition wie folgt angibt:

```
%token  IDENTIFIER

%left   '+' '-'
%left   '*' '/'

%%
```

%left vereinbart eine Liste von Eingabesymbolen als links-assoziativ und mit gleichem Vorrang zueinander. Der Vorrang nimmt dann zu in der Reihenfolge weiterer **%left**-, **%right**- und **%nonassoc**-Vereinbarungen in der Eingabe zu *yacc*. Mit **%right** vereinbart man natürlich rechts-assoziative Eingabesymbole. **%nonassoc** ordnet Eingabesymbolen untereinander gleichen Vorrang zu, verbietet aber, daß sie in einer Folge auftreten.

Diese Kombination von Mehrdeutigkeit in der Grammatik und expliziten Vorschriften wird bevorzugt. Dadurch benötigt man weniger Grammatikbegriffe und man vereinfacht die Grammatik.

1.6 Beispiel

Dieses Buch enthält einen vollständigen Compiler für *kleinC*, eine rudimentäre Untermenge von C. Die verschiedenen Module des Compilers werden jeweils in einem 'Beispiel'-Abschnitt gegen Ende jedes Kapitels beschrieben.

Hier ist unsere informelle Definition von *kleinC* in EBNF. Wir umgeben Eingabesymbole mit Anführungszeichen, wenn sie aus Sonderzeichen bestehen. Reservierte Worte sind in Großbuchstaben geschrieben. **Identifier** und **Constant** sind ebenfalls Eingabesymbole; im Gegensatz zu den anderen Eingabesymbolen repräsentieren diese jedoch Klassen von Worten und nicht etwa sich selbst. Die Definition ist absichtlich etwas weitschweifig, um Erweiterungen zu vereinfachen, die dem Leser als Aufgaben überlassen werden.

```
program
        : definition { definition }

definition
        : data_definition
        | function_definition

data_definition
        : INT declarator { ',' declarator } ';'

declarator
        : Identifier

function_definition
        : [ INT ] function_header function_body

function_header
        : declarator parameter_list

parameter_list
        : '(' [ Identifier_list ] ')' { parameter_declaration }

Identifier_list
        : Identifier { ',' Identifier }

parameter_declaration
        : INT declarator { ',' declarator } ';'

function_body
        : '{' { data_definition } { statement } '}'
```

```
statement
        : [ expression ] ';'
        | '{' { data_definition } { statement } '}'
        | IF '(' expression ')' statement [ ELSE statement ]
        | WHILE '(' expression ')' statement
        | BREAK ';'
        | CONTINUE ';'
        | RETURN [ expression ] ';'

expression
        : binary { ',' binary }

binary
        : Identifier '='  binary                    von rechts
        | Identifier '+=' binary
        | Identifier '-=' binary
        | Identifier '*=' binary
        | Identifier '/=' binary
        | Identifier '%=' binary
        | binary '==' binary    Vorrang             von links
        | binary '!=' binary
        | binary '<'  binary    Vorrang
        | binary '<=' binary
        | binary '>'  binary
        | binary '>=' binary
        | binary '+'  binary    Vorrang
        | binary '-'  binary
        | binary '*'  binary    Vorrang
        | binary '/'  binary
        | binary '%'  binary
        | unary

unary
        : '++' Identifier
        | '--' Identifier
        | primary

primary
        : Identifier
        | Constant
        | '(' expression ')'
        | Identifier '(' [ argument_list ] ')'

argument_list
        : binary { ',' binary }
```

Die Semantik von *kleinC* ergibt sich aus der Semantik von C, mit entsprechenden Auslassungen. Die Sprache hat nur den **int**-Datentyp, Funktionen mit einer variablen und beliebigen Anzahl von Parametern und mit **int**-Resultat, skalare Variablen, die global, lokal zu einer Funktion oder zu einem Block definiert werden können sowie die Kontrollstrukturen **if-else** und **while**. Eine Reihe von C-Operatoren ist verfügbar. Nur eine Datei soll auf einmal übersetzt werden; diese Datei muß eine **main()**-Funktion enthalten. Diese letzte Bedingung müssen wir gesondert behandeln (siehe Kapitel 5).

Die informelle Definition wird in BNF übertragen. Dazu muß man hauptsächlich die Wiederholungen mit Hilfe von Rekursion ausdrücken. Eine Liste von **%token**-Vereinbarungen und eine Vorrangtabelle muß auch noch hinzugefügt werden. Die folgende Definition wird von *yacc* akzeptiert:

```
/*
 *      kleinC
 *      Syntaxanalyse
 *      (s/r Konflikt: bei ELSE)
 */

/*
 *      Eingabesymbole
 */

%token  Identifier
%token  Constant
%token  INT
%token  IF
%token  ELSE
%token  WHILE
%token  BREAK
%token  CONTINUE
%token  RETURN
%token  ';'
%token  '('
%token  ')'
%token  '{'
%token  '}'
%token  '+'
%token  '-'
%token  '*'
%token  '/'
%token  '%'
%token  '>'
%token  '<'
```

```
%token  GE      /* >= */
%token  LE      /* <= */
%token  EQ      /* == */
%token  NE      /* != */
%token  '&'
%token  '^'
%token  '|'
%token  '='
%token  PE      /* += */
%token  ME      /* -= */
%token  TE      /* *= */
%token  DE      /* /= */
%token  RE      /* %= */
%token  PP      /* ++ */
%token  MM      /* -- */
%token  ','

/*
 *      Vorrangtabelle
 */

%right  '=' PE ME TE DE RE
%left   '|'
%left   '^'
%left   '&'
%left   EQ NE
%left   '<' '>' GE LE
%left   '+' '-'
%left   '*' '/' '%'
%right  PP MM

%%

program
        : definitions

definitions
        : definition
        | definitions definition

definition
        : function_definition
        | INT function_definition
        | declaration
```

```
function_definition
        : Identifier '(' optional_parameter_list ')'
          parameter_declarations compound_statement

optional_parameter_list
        : /* keine Parameter */
        | parameter_list

parameter_list
        : Identifier
        | parameter_list ',' Identifier

parameter_declarations
        : /* null */
        | parameter_declarations parameter_declaration

parameter_declaration
        : INT parameter_declarator_list ';'

parameter_declarator_list
        : Identifier
        | parameter_declarator_list ',' Identifier

compound_statement
        : '{' declarations statements '}'

declarations
        : /* null */
        | declarations declaration

declaration
        : INT declarator_list ';'

declarator_list
        : Identifier
        | declarator_list ',' Identifier

statements
        : /* null */
        | statements statement
```

```
statement
        : expression ';'
        | ';'    /* leere Anweisung */
        | BREAK ';'
        | CONTINUE ';'
        | RETURN ';'
        | RETURN expression ';'
        | compound_statement
        | if_prefix statement
        | if_prefix statement ELSE statement
        | loop_prefix statement

if_prefix
        : IF '(' expression ')'

loop_prefix
        : WHILE '(' expression ')'

expression
        : binary
        | expression ',' binary

binary
        : Identifier
        | Constant
        | '(' expression ')'
        | Identifier '(' optional_argument_list ')'
        | PP Identifier
        | MM Identifier
        | binary '+' binary
        | binary '-' binary
        | binary '*' binary
        | binary '/' binary
        | binary '%' binary
        | binary '>' binary
        | binary '<' binary
        | binary GE binary
        | binary LE binary
        | binary EQ binary
        | binary NE binary
        | binary '&' binary
        | binary '^' binary
        | binary '|' binary
        | Identifier '=' binary
        | Identifier PE binary
```

```
        | Identifier ME binary
        | Identifier TE binary
        | Identifier DE binary
        | Identifier RE binary

optional_argument_list
        : /* keine Argumente */
        | argument_list

argument_list
        : binary
        | argument_list ',' binary
```

1.7 Zur Typographie

Wie schon erwähnt wurde, erfolgt die Eingabe für *yacc* praktisch formatfrei; einige Konventionen haben sich jedoch als hilfreich erwiesen:

Jeder Grammatikbegriff sollte nur einmal auf der linken Seite einer Regel auftreten. Wir schreiben den Grammatikbegriff allein auf einer Zeile und zwar ganz am linken Rand. Alle Formulierungen werden dann nacheinander angegeben.

Der Doppelpunkt zwischen linker und rechter Seite einer Regel wird eine Tabulatorposition eingerückt, die erste Formulierung wird ein Leerzeichen nach dem Doppelpunkt mehr eingerückt und Aktionen - denen wir in Kapitel 3 begegnen - werden schließlich zwei Tabulatorpositionen eingerückt.

Doppelpunkt, der Strich |, der alternative Formulierungen trennt sowie das Semikolon am Ende einer Regel werden alle vertikal untereinander ausgerichtet.

Aus Platzgründen haben wir in diesem Buch zwar am Schluß der Regeln kein Semikolon angegeben. *yacc* akzeptiert die Grammatik trotzdem, aber in der Praxis sollte man das Semikolon unbedingt angeben. Läßt man es weg, werden Tippfehler leicht übersehen, etwa wenn man versehentlich einen Strich an Stelle eines Doppelpunkts schreibt.

Zwischen zwei Regeln befindet sich immer eine leere Zeile.

Leere Formulierungen enthalten immer einen entsprechenden Kommentar - dies ist zur Fehlersuche sehr hilfreich.

Wir bevorzugen lange Namen, die wir mit Unterstrichen gliedern. Zu diesem Zweck können auch Punkte verwendet werden. Eine durch Komma unterteilte Liste signalisieren wir auch meistens mit der Nachsilbe **_list**. Plural ist ein Hinweis, daß ein Grammatikbegriff eine Folge von Objekten beschreibt.

1.8 Aufgaben

1. Wählen Sie eine *kleine* Untermenge von Pascal und beschreiben Sie sie in BNF. Fügen sie entsprechende **%token**-Vereinbarungen und eine Vorrangtabelle hinzu. Prüfen Sie diese Sprachdefinition mit Hilfe von *yacc* und korrigieren Sie sie falls nötig, bis sie von *yacc* akzeptiert wird (das heißt, bis *yacc* nur noch unvermeidbare *shift/reduce*-Konflikte bemängelt.) ✏

2. Erweitern Sie die Definition von *kleinC* um einen weiteren Aspekt von C, zum Beispiel um Vektoren. Ändern Sie die Grammatik im Abschnitt 1.6 so, daß die neue Beschreibung von *kleinC* ebenfalls von *yacc* akzeptiert wird. ✏

3. Schreiben Sie eine Grammatik für EBNF in BNF. Eine Grammatik für EBNF in EBNF befindet sich im Abschnitt 1.2. Wie muß man diese Grammatik für BNF ändern? Fügen Sie **%token**-Anweisungen und was sonst nötig ist hinzu, damit die Grammatik von *yacc* akzeptiert wird. Hinweis: um alle *shift/reduce*-Konflikte zu entfernen, werden Sie wohl ein Semikolon am Ende jeder Regel verlangen. ✏

Kapitel 2: Worterkennung

2.1 Zweck

Wir haben im letzten Kapitel gesehen, daß eine Sprache eine Menge von Sätzen ist, die ihrerseits Folgen von Eingabesymbolen sind. Normalerweise hat eine Programmiersprache drei Arten von Eingabesymbolen: Operatoren werden als (kurze) Folgen von Sonderzeichen dargestellt, reservierte Worte werden als vordefinierte Folgen von Buchstaben dargestellt, deren Bedeutung nicht verändert werden kann und vom Benutzer eingeführte Eingabesymbole schließlich sind Konstanten und Namen mit einer speziellen Syntax. Dann gibt es natürlich noch Zwischenraum (Leerzeichen, Tabulatorzeichen und Zeilentrenner). In den meisten Programmiersprachen trennen Zwischenraumzeichen nur die Eingabesymbole gegeneinander ab, sind aber sonst unwesentlich. Auch für Kommentare gibt es entsprechende syntaktische Bestimmungen, die in fast allen Programmiersprachen [Wic73] verschieden sind, aber sonst genauso unbedeutend wie Zwischenraum.

Dieses Kapitel behandelt die *lexikalische Analyse*, den Teil des Compilers, der Eingabesymbole aus der unstrukturierten Folge von Zeichen konstruiert, die der Compiler als eigentliche Eingabe erhält. Zwischenraum und Kommentare werden normalerweise ignoriert, Operatoren und reservierte Worte müssen erkannt und mit Hilfe einer internen Repräsentierung weitergeleitet werden - normalerweise verwendet man dazu kleine Integer-Konstanten. Konstanten und Namen speichert man in einer geeigneten Tabelle und leitet einen Gattungsbegriff wie **Identifier** oder **Constant** zusammen mit einem Verweis auf den Eintrag in der Tabelle weiter.

Wir könnten das Problem der lexikalischen Analyse im Zusammenhang mit der eigentlichen Sprachdefinition folgendermaßen lösen:

```
statement
        : 'I' 'F' condition 'T' 'H' 'E' 'N' statement
```

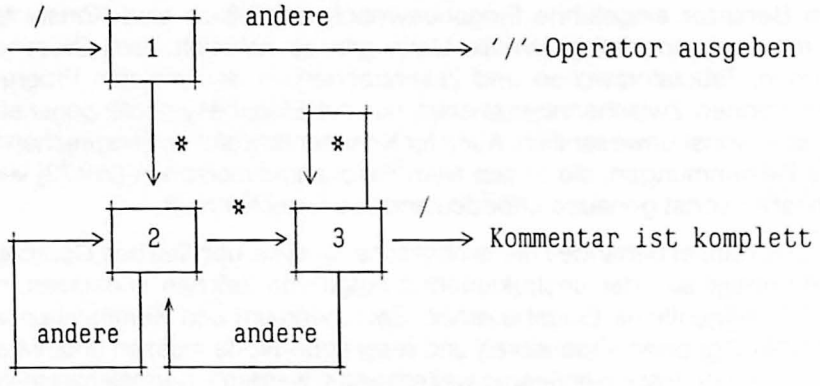
Dadurch entsteht aber ganz bestimmt eine Grammatik mit einer Vielzahl von Konflikten, und die Technik ist außerdem extrem ineffizient.

Die lexikalische Analyse verursacht einen großen Teil des Rechenaufwands in einem Compiler. Wenn wir die lexikalische Analyse weitgehend unabhängig vom Rest des Compilers entwickeln, können wir geeignetere Techniken verwenden und gleichzeitig in einem einzigen Modul die gesamte Information darüber verbergen, wie unsere Programmiersprache in einem bestimmten Zeichensatz repräsentiert wird.

2.2 Werkzeuge zur Definition

Wie fassen wir nun wirklich einzelne Zeichen zu Eingabesymbolen zusammen? Lexikalische Analyse ist der klassische Anwendungsbereich der Theorie der endlichen Automaten. Eine Übergangsmatrix für die lexikalische Definition einer Pro-

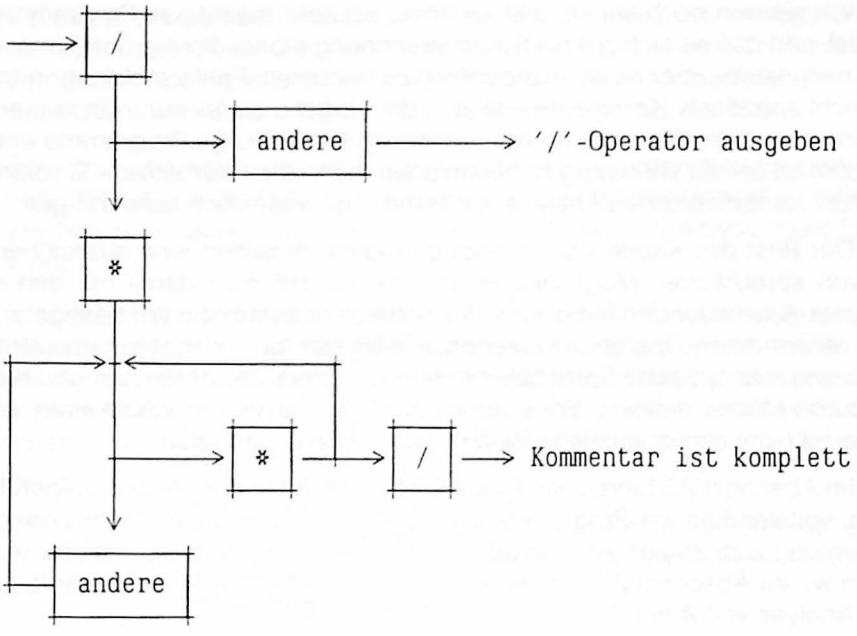
grammiersprache kann man im allgemeinen leicht entwerfen, und in ein bis zwei Tagen hat man 'auf die Schnelle' praktisch jede solche Übergangsmatrix in ein entsprechendes Programm verwandelt. Problematisch sind dabei in der Regel nur Zeichenketten, Kommentare und (Gleitkomma-) Konstanten, etwa in dieser Reihenfolge mit zunehmendem Schwierigkeitsgrad. Zur Illustration betrachten wir einen Kommentar im Stil von C, dargestellt als Zustandsdiagramm:



In einem Zustandsdiagramm sind die Zustände die numerierten Knoten eines orientierten Graphen, und die möglichen Übergänge sind die Kanten, markiert mit den Eingabezeichen, die die jeweilige Zustandsänderung verursachen. Typischerweise ist der Ausstieg aus einem solchen Diagramm, wenn ein Eingabesymbol erkannt und die Zeichen seiner Repräsentierung zusammengefaßt wurden, nicht immer wohldefiniert. Manchmal gehört das nächste Eingabezeichen zum nächsten Symbol, manchmal wurde es noch gar nicht eingelesen.

Ein Zustandsdiagramm ist eng mit einem Syntaxgraphen verwandt. Im Zustandsdiagramm beschreiben die Kanten die Übergänge. Im Syntaxgraphen enthalten die Knoten die Symbole, die man finden muß um weiterzukommen. Der hauptsächliche Unterschied ist, daß Syntaxgraphen sich häufig gegenseitig aufrufen, während diese einfachen Zustandsdiagramme dies nicht tun sollen.

Ein Vergleich der zwei Diagramme läßt vermuten, warum man Zustandsdiagramme zum Entwurf der lexikalischen Analyse bevorzugt. Die Technik als solche ist aber fehleranfällig, und das Resultat ist meist ziemlich schwer zu ändern.



Direkt aus der Theorie kann man eine bessere Lösung zum Problem der lexikalischen Analyse entwickeln: wir brauchen eine bequeme Spezifikationssprache für die lexikalische Definition unserer Sprache, wir brauchen einen Übersetzer, der aus einer solchen Spezifikation geeignete Tabellen konstruiert, und wir brauchen schließlich einen Interpreter, der den endlichen Automaten anhand der Tabellen simuliert.

Einen derartigen Übersetzer gibt es tatsächlich: *lex* [Les78b]. *lex* akzeptiert eine Tabelle mit Mustern, die an die im Editor verwendeten Muster erinnern,[1] und produziert ein tabellengetriebenes C-Programm, das in der Lage ist, Folgen von Eingabezeichen zu erkennen, die den Mustern genügen. Wie das *substitute*-Kommando im Editor *ed* zeigt, sind Muster eine bequeme Schreibweise zur Erkennung bestimmter Zeichenfolgen.

Muster sind eine sehr zweckmäßige Spezifikationssprache für den endlichen Automaten, der tatsächlich konstruiert wird. Jedem Muster kann man eine C-Anweisung zuordnen, die ausgeführt wird, wenn eine Folge von Eingabezeichen gefunden wurde, die dem Muster genügt. Bei einfachen Wort-Übersetzern gibt man mit der C-Anweisung normalerweise eine leicht modifizierte Version der Eingabezeichen als Standard-Ausgabe aus. Zur Weiterverarbeitung in einem Compiler sollte im Rahmen der C-Anweisung mit **return** dem Aufrufer der Funktion zur lexikalischen Analyse die Eingabefolge geeignet codiert übergeben werden.

[1] Siehe **ed(1)** in [Ker78a].

Wir werden noch sehen, daß *lex* allein ein sehr mächtiges Programmierwerkzeug ist, und daß es sich gut zur Spracherkennung eignet. Wenigstens unserer Meinung nach ist *lex* aber recht unangenehm zu benutzen: Fehlermeldungen sind kurz und nicht spezifisch, Kommentare sind in der Eingabe zu *lex* nur mühsam anzubringen und bei naiver Verwendung von *lex* können leicht riesige Programme entstehen. Trotzdem ist *lex* als Werkzeug zu bevorzugen, denn die Alternative - Codierung der Funktion zur lexikalischen Analyse von Hand - ist wesentlich aufwendiger.

Der Rest des Kapitels beschreibt genügend detailliert eine ausreichende Anzahl von sprachlichen Möglichkeiten bei *lex*, so daß man damit mit den meisten Compiler-Anwendungen fertig wird. Wir betrachten zuerst die am häufigsten in Mustern verwendeten Operationen, sehen, wie Muster für *lex* angegeben werden und lernen, wie man typische Sprachelemente wie Namen, Zeichenketten und Kommentare durch Muster definiert. Wir erklären zwar die meisten Möglichkeiten, aber das Kapitel ist nicht als umfassende Beschreibung von *lex* gedacht.

Im Abschnitt 2.5 führen wir *lex* als Programmgenerator vor und zeigen ein paar kleine, vollständige *lex*-Programme zum Einfügen von Dateien, zum Zerlegen von Dateien und zum Zählen von Worten in einem Text. Unsere *kleinC*-Implementierung setzen wir im Abschnitt 2.7 fort, wo wir dann die komplette Funktion für die lexikalische Analyse vorführen.

2.3 Muster

Benutzer von *ed* und ähnlichen Texteditoren kennen bereits folgende Bausteine für Muster:

Ziffern, Buchstaben und manche Sonderzeichen repräsentieren sich selbst.

Ein Punkt steht für ein beliebiges Zeichen, mit Ausnahme des Zeilentrenners.

Eckige Klammern, [und], umgeben eine Folge von Zeichen, die als *Zeichenklasse* bezeichnet werden soll. Die Zeichenklasse repräsentiert jedes einzelne ihrer Mitglieder, oder auch ein einzelnes Zeichen, das *nicht* in der Zeichenklasse vorkommt, wenn die definierende Zeichenfolge mit ^ beginnt. Innerhalb der Zeichenfolge bezeichnet – zwischen zwei Zeichen den gesamten Bereich mit Einschluß dieser zwei Zeichen.

Folgt * einem dieser Bausteine im Muster, kann die entsprechende Eingabe beliebig oft oder auch gar nicht erfolgen.

^ am Anfang eines Musters repräsentiert den Anfang einer Eingabezeile.

$ am Ende eines Musters repräsentiert das Ende einer Eingabezeile (aber nicht das Zeilentrennzeichen selbst).

Mit einer entsprechenden Ersatzdarstellung für Sonderzeichen und Zwischenraum können wir jetzt schon die meisten Eingabesymbole einer Programmiersprache mit Mustern darstellen. Es gibt zwei Möglichkeiten zur Ersatzdarstellung, also zur Darstellung von Zeichen, die in Mustern für *lex* normalerweise Sonderbedeutung besitzen:

\ zitiert ein einzelnes, unmittelbar folgendes Sonderzeichen. Insbesonders repräsentieren zwei \-Zeichen nacheinander ein einzelnes \-Zeichen. \ kann den Buchstaben **b**, **n** und **t** vorausgehen; die Kombinationen stehen dann für *backspace*, *Zeilentrenner* und *Tabulatorzeichen*, genau wie in C.

Eine bessere Methode der Ersatzdarstellung, insbesondere für eine Folge von Zeichen, besteht darin, eine Reihe von Zeichen mit Doppel-Anführungszeichen zu umgeben. Die Zeichen verlieren dadurch ihre besondere Bedeutung.

Die meisten Sonderzeichen haben für *lex* irgendeine Sonderbedeutung; soll sich ein Sonderzeichen selbst darstellen, so zitiert man es am besten immer. Innerhalb einer Zeichenklasse braucht man keine Ersatzdarstellung für Sonderzeichen. Zwischenraum muß immer in Doppel-Anführungszeichen oder mit Hilfe von \ dargestellt werden. Innerhalb von Doppel-Anführungszeichen stellt man ein Doppel-Anführungszeichen durch \" dar, genau wie in C.

Als Beispiel betrachten wir Muster für einige Eingabesymbole in *kleinC*. Operatoren werden einfach zitiert - wir bevorzugen im allgemeinen Doppel-Anführungszeichen vor der \-Konvention:

```
"*"
"--"
"*="
```

und so weiter. Wir brauchen ein Muster für Zwischenraum:

```
[ \t\n]
```

Constant kann aus einer Folge von Ziffern bestehen:

```
[0-9][0-9]*
```

Identifier folgt den Regeln von C, das heißt, ein Name beginnt mit einem Buchstaben oder mit _ und er kann dann beliebig viele Buchstaben, _-Zeichen oder Ziffern enthalten:

```
[A-Za-z_][A-Za-z0-9_]*
```

Das nächste Muster versucht, einen Kommentar innerhalb einer Zeile zu erkennen:

```
"/*".*"*/"
```

Ein Kommentar soll mit /* beginnen und mit dem *ersten* Auftreten von */ beendet werden. Ein Kommentar kann sich über viele Zeilen erstrecken.

Die letzten zwei Muster zeigen gewisse Grenzen auf, die gelegentlichen Benutzern von *ed* gut bekannt sein sollten: nach Definition repräsentiert ein Muster die *längste* mögliche Folge von Eingabezeichen. Bei Konstanten und Namen beabsichtigen wir dies wirklich, aber bei Kommentaren sollten wir die Muster genau umge-

kehrt interpretieren! Fraglich ist übrigens auch, was passiert, wenn wir folgendes angeben:

```
"<"
"<="
```

also zwei Muster, die am Anfang die gleiche Folge von Zeichen repräsentieren oder gar

```
"int"
[a-z][a-z]*
```

also zwei Muster, von denen das eine eine Untermenge der Eingabesymbole des andern Musters repräsentiert. Wir werden uns im nächsten Abschnitt mit solchen mehrdeutigen Mustern befassen.

Um erfolgreich mit C- und Pascal-Kommentaren und Zeichenketten verfahren zu können, müssen wir noch ein paar Muster-Bausteine von *lex* vorstellen. Wie bei *egrep*[1] kann man auch bei *lex* Klammern zur Gruppierung in einem Muster verwenden, **|** trennt Alternativen, **+** erlaubt, daß der vorhergehende Baustein ein- oder mehrmals verwendet wird und **?** bezeichnet ein optionales Auftreten eines Bausteins. Die letzten zwei Möglichkeiten vereinfachen einige der früher gezeigten Muster, zum Beispiel kann man Integer-Konstanten mit folgendem Muster erkennen:

```
[0-9]+
```

Ein Eingabesymbol, das durch einzelne Zeichen begrenzt wird wie etwa ein Kommentar in Pascal, ist einfach zu beschreiben:

```
"{"[^}]*"}"
```

Mit Hilfe von Alternativen kann man die Technik leicht auf Pascal- oder C-Zeichenketten ausdehnen, bei denen das umgebende Zeichen verdoppelt oder anders dargestellt werden muß, wenn es innerhalb der Zeichenkette vorkommen soll:

```
\'([^'\n]|\'\')+\'
\"([^"\n]|\\["\n])*\"
```

Man beachte, daß die hier definierte Zeichenkette im Stil von Pascal sich nicht über mehrere Zeilen erstrecken kann und daß sie wenigstens ein Zeichen enthalten muß. Eine C-Zeichenkette kann sich über mehrere Zeilen erstrecken, wenn der Zeilentrenner mit Hilfe von \ markiert wird, die Zeichenkette kann mit \ zitierte Doppel-Anführungszeichen enthalten und sie kann auch leer sein.

Wird zur Begrenzung eines Symbols eine Folge von mehreren Zeichen verwendet, wie etwa bei Kommentaren in C, die mit /* und */ umgeben sein müssen, oder bei Kommentaren in Pascal mit (* und *), so wird das entsprechende Muster recht kompliziert. Die Grundidee ist, die abschließenden Zeichen nicht mehr zu er-

[1] Siehe **grep(1)** in [Ker78a].

lauben nachdem man die Zeichen am Anfang des Objekts erkannt hat; dies läßt sich leider nur durch eine Aufzählung geschlossen darstellen. Hier ist ein erster Vorschlag für ein Muster, das einen Kommentar in C erkennen soll:

```
"/*"([^*/]|[^*]"/"|"*"[^/])*"*/"
```

Der Kommentar beginnt mit /* und endet mit */. Dazwischen kann eine von drei Alternativen beliebig oft vorkommen, nämlich entweder jedes Zeichen außer * und /, oder ein /-Zeichen mit einem beliebigen Zeichen aber nicht gerade * davor, oder schließlich ein *-Zeichen, dem alles nur nicht / folgt. Man beachte, daß eine Zeichenklasse mit dem Komplementoperator ^ ein gefährliches Instrument ist - normalerweise gehört dann der Zeilentrenner zur Zeichenklasse, und damit kann eine unvorsichtige Wiederholung der Zeichenklasse leicht eine ganze Eingabedatei verschlucken!

Leider ist unsere 'Lösung' für Kommentare in C noch nicht ganz perfekt: /*/ wird nicht als Anfang eines Kommentars erkannt, da in dem vorgeschlagenen Muster / erst einem Zeichen im Kommentar folgen kann und /***/ wird nicht als vollständiger Kommentar erkannt, da in dem Muster * nur einem Zeichen im Kommentar vorausgehen kann; damit wird das dritte *-Zeichen als Teil der Begrenzung ignoriert. Eine bessere Lösung ist das folgende Muster, das leider noch schwieriger verständlich ist:

```
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"
```

Jetzt können /* am Anfang beliebig viele / folgen (auch keine), und ebenso können eine Reihe von * noch dem abschließenden */ vorausgehen. Damit sind die eben besprochenen extremen Fälle auch erfaßt.

lex-Muster können noch interessanter gestaltet werden. Eine Wiederholung kann auf eine bestimmte Anzahl eingeschränkt werden, mit einer Schreibweise wie

```
[a-z][a-z0-9]{0,7}
```

Dieses Muster beschreibt einen Namen, der mit einem Kleinbuchstaben beginnt, dem bis zu sieben Buchstaben oder Ziffern folgen.

Ein Muster kann auf einen bestimmten rechten Kontext eingeschränkt werden, der selbst durch das Muster nicht repräsentiert wird. Der gewünschte rechte Kontext wird im Anschluß an / nach dem Muster angegeben:

```
-0[xX]/[0-9a-f]+
```

würde Vorzeichen und Basis einer negativen hexadezimalen Konstante in C erkennen, ohne dabei aber die Ziffern der Konstante selbst zu erfassen.

Vor Mustern können Bedingungen angegeben werden. Dadurch kann man im wesentlichen verschiedene Gruppen von Mustern einzeln dynamisch auswählen, auch wenn sie zusammen definiert wurden.

Ein sehr fähiger (und entsprechend vorsichtiger) Programmierer kann schließlich auch die Funktionen direkt manipulieren, die der von *lex* generierte Automat für Eingabe, Ausgabe und Puffer-Management verwendet. Den geduldigen und einfallsreichen Leser verweisen wir mit Freuden auf die Originalliteratur [Les78b] sowie auf [Sch85].

2.4 Mehrdeutigkeit als Tugend

Eine Liste von *lex*-Mustern ist im allgemeinen höchst mehrdeutig. *lex* verwendet zwei Regeln um Ordnung zu schaffen:

> *lex* benutzt immer das Muster, das die längstmögliche Folge von Eingabezeichen repräsentiert.
>
> Repräsentieren dabei immer noch zwei Muster die gleiche Eingabe, so wird das erste Muster in der für *lex* angegebenen Tabelle von Mustern verwendet.

Beide Regeln sind in der Praxis von Vorteil und nicht etwa Fallstricke. Die erste Regel sorgt dafür, daß durch die Muster

```
int
[a-z]+
```

integer als Instanz des zweiten Musters komplett erkannt wird, und nicht etwa nur die Zeichenfolge **int** am Anfang des längeren Worts. Die zweite Regel sorgt hier dafür, daß das zuerst angegebene Muster **int** und nicht das zweite, allgemeinere Muster eine isolierte Folge **int** in der Eingabe erkennt.

Wie im vorigen Abschnitt demonstriert wurde, kann die erste Regel gegen Mehrdeutigkeit ins Auge gehen, wenn sie sorglos benutzt wird. Die zweite Regel führt zu einer Anordnung, bei der das allgemeinste Muster, etwa für einen **Identifier**, *zuletzt* angegeben wird, wobei speziellere Muster davor dann Ausnahmen erkennen.

Betrachten wir folgendes Erkennungsproblem: wenn man deutsche Texte auf einer amerikanischen Tastatur tippt, tippt man Umlaute am schnellsten als **ae**, **oe**, **ue**, usw. Bevor man nun das resultierende Dokument mit *nroff* oder *troff* bearbeitet, etwa mit dem *ms*-Makropaket [Les78a], sollte man alle diese Buchstabenkombinationen besser mit dem in *ms* definierten String ***:** darstellen, also etwa als ***:a**, ***:o**, usw. Das Muster

```
ue
```

erkennt aber die Buchstabenkombination **ue** auch in Worten wie **Quelle** oder **eventuell**, und in diesen Worten sollte später kein Umlaut auftauchen! Wir müssen also diese Sonderfälle abfangen *bevor* wir die allgemeine Kombination angeben:

```
[Qq]ue
ntue
ue
```

Da *lex* durchaus mehrdeutige Tabellen von Mustern als Eingabe akzeptiert, können wir weitere Sonderfälle leicht hinzufügen, wenn sie uns begegnen.

Anders als bei *yacc*, wo nur bei *reduce/reduce*-Konflikten die Reihenfolge der Spezifikationen eine Rolle spielt, ist die Reihenfolge der Muster bei *lex* sehr wichtig, da Konflikte im Sinne von *yacc* sehr häufig auftreten. Dies führt leider oft zu einer notwendigen, unlogischen Anordnung der Muster. Die Reihenfolge der Muster spielt die Rolle einer Kontrollstruktur.

2.5 "lex"-Programme

Die Eingabe für *lex* erfolgt in einer Datei. Sie hat drei Teile, die durch Zeilen mit **%%** in den ersten beiden Positionen getrennt werden:

```
erster Teil

%%

Muster          Aktion
        ...

%%

dritter Teil
```

Der erste Teil ist optional. Dort stehen Zeilen, die die Dimensionierung bestimmter interner Tabellen in *lex* kontrollieren sowie Definitionen für Textersatz, wie wir im Abschnitt 2.7 noch sehen werden. Im ersten Teil kann auch globaler C-Programmtext stehen, der dann von zwei Zeilen umgeben sein muß, die mit **%{** beziehungsweise **%}** beginnen. Auch wenn im ersten Teil der *lex*-Spezifikation nichts angegeben wird, darf der Trenner **%%** zwischen dem ersten und zweiten Teil der Spezifikation nicht fehlen.

Der dritte Teil und der vorausgehende Trenner sind optional. Im dritten Teil kann C-Programmtext stehen, der unverändert in die Ausgabe übernommen wird. Wie wir noch sehen werden, stehen dort normalerweise die (lokalen) Funktionen, die im zweiten Teil verwendet werden.

Der zweite Teil der Eingabe für *lex* besteht aus einer Tabelle von Mustern und Aktionen. Dieser Teil ist ziemlich zeilenorientiert. Ein Muster beginnt am Anfang einer Zeile und reicht von dort zum ersten Zwischenraumzeichen, das nicht zitiert ist. Im Anschluß an beliebig viel Zwischenraum wird dann eine Aktion angegeben, die dem Muster zugeordnet werden soll. Die Aktion ist eine einzige C-Anweisung, oder eine Folge von Anweisungen, die mit geschweiften Klammern umgeben ist. Als Aktion kann auch nur das Zeichen **|** angegeben werden, das anzeigt, daß das aktuelle Muster die gleiche Aktion verwendet wie das folgende Muster.

lex hat selbst keine Vorkehrungen für Kommentare(!), aber innerhalb von geschweiften Klammern können normale C-Kommentare verwendet werden.

Aus der Tabelle konstruiert *lex* eine Funktion **yylex()**, die in der Datei *lex.yy.c* abgelegt wird. Montiert man diese Funktion mit einem Programm und ruft sie auf, wird die Standard-Eingabe gelesen, bis die nächste, längstmögliche Folge von Eingabezeichen gesammelt wurde, die durch ein Muster repräsentiert werden kann. Anschließend wird die diesem Muster zugeordnete Aktion ausgeführt. Enthält diese Aktion eine **return**-Anweisung, wird der Aufruf von **yylex()** dadurch beendet, möglicherweise mit einem Resultatwert der durch die **return**-Anweisung bestimmt wird.

lex fügt zu den vom Benutzer angegebenen Mustern noch ein Muster mit Aktion dergestalt hinzu, daß alle nicht anders erkannten Eingabezeichen in die Standard-Ausgabe kopiert werden.

Das folgende *lex*-Programm entfernt alle Großbuchstaben aus seiner Eingabe:

```
%{
/*
 *      Grossbuchstaben entfernen
 */
%}

%%

[A-Z]+          ;
```

Angenommen, daß sich dieses Programm in der Datei *exuc.l* befindet, dann produzieren folgende Kommandos ein ausführbares Programm namens *exuc*:

```
lex exuc.l
cc lex.yy.c -ll -o exuc
```

-ll verweist auf die *lex*-Bücherei, die unter anderem eine **main()**-Funktion enthält, die **yylex()** einmal aufruft. Diese Bücherei muß immer angegeben werden, wenn eine von *lex* konstruierte Funktion **yylex()** montiert werden soll.

Eine nützlichere Aktion benötigt Zugriff auf die Folge von Eingabezeichen, die ihr zugehöriges Muster erkannt hat.

```
extern char yytext[];
```

enthält diese Zeichenfolge, wie üblich mit einem Nullzeichen abgeschlossen;

```
extern int yyleng;
```

hat den Wert **strlen(yytext)** und

```
extern int yylineno;
```

enthält die Nummer der aktuellen Eingabezeile. Betrachten wir einige relativ sinnvolle Programme:

```
%{
/*
 *      Zeilen numerieren
 */
%}

%%

\n      ECHO;
^.*$    printf("%d\t%s", yylineno, yytext);
```

Dieses erste Programm gibt seine Standard-Eingabe aus, wobei vor jeder nicht-leeren Eingabezeile die Zeilennummer und ein Tabulatorzeichen ausgegeben werden. **ECHO** ist im C-Programm von *lex* definiert worden; diese Aktion sorgt dafür, daß **yytext[]** ausgegeben wird.

Wenn wir auch die leeren Zeilen numerieren wollen, kann das folgende Programm verwendet werden:

```
%{
/*
 *      Zeilen numerieren
 */
%}

%%

^.*\n   printf("%d\t%s", yylineno-1, yytext);
```

Jetzt erkennen wir den Zeilentrenner als Teil des Musters. **yylineno** wurde schon auf die Folgezeile hochgezählt, wenn unsere Aktion ausgeführt wird.

Als nächstes folgt das liebste Dienstprogramm eines Autors:

```
%{
/*
 *      Worte zaehlen
 */

int     nchar, nword, nline;
%}
```

```
%%

\n              ++ nchar, ++ nline;
[^ \t\n]+       ++ nword, nchar += yyleng;
.               ++ nchar;

%%

main()
{
        yylex();
        printf("%d\t%d\t%d\n", nchar, nword, nline);
}
```

Dieses Beispiel illustriert, was man im dritten Teil einer *lex*-Definition angeben kann: wir verwenden hier eine eigene **main()**-Funktion, die hier die beim Aufruf von **yylex()** gesammelte Statistik ausgibt.

Das nächste Beispiel ist typisch für eine große Zahl von Problemen, bei denen eine bestimmte Aktion erfolgen muß, wenn ein bestimmtes Muster erkannt wird. Hier ist es das Einfügen einer Datei. Der größte Teil der Eingabe wird einfach kopiert. Ähnliche Anwendungen sind zum Beispiel die graphische Verbrämung von reservierten Worten in einem Programmtext für Publikationszwecke, das Sammeln eines Inhaltsverzeichnisses, usw.

```
%{
/*
 *      .so Dateiname   Datei einfuegen
 */

#include <ctype.h>
#include <stdio.h>

static include();
%}


%%

^".so".*\n      { yytext[yyleng-1] = '\0'; include(yytext+3); }

%%
```

```
static include(s)
        char * s;
{       FILE * fp;
        int i;
        while (*s && isspace(*s))
                ++ s;
        if (fp = fopen(s, "r"))
        {       while ((i = getc(fp)) != EOF)
                        output(i);
                fclose(fp);
        }
        else
                perror(s);
}
```

Die **output()**-Funktion verwendet *lex* für alle Ausgaben. Unser Programm wird nur mit einer Ebene von Kommandos zur Einfügung von Dateien fertig.

Das letzte Beispiel verteilt eine Ausgabe auf verschiedene Dateien, die durch Angaben in der Eingabe ausgewählt werden. Wir haben hier eine Variante des *split*-Kommandos vor uns.

```
%{
/*
 *      .di Dateiname   Datei zerlegen
 */

#include <ctype.h>
#include <stdio.h>
static divert();
%}

%%

^".di".*\n      { yytext[yyleng-1] = '\0'; divert(yytext+3); }

%%

static divert(s)
        char * s;
{
        while (*s && isspace(*s))
                ++ s;
        if (! freopen(s, "w", stdout))
                perror(s);
}
```

Mit Dienstprogrammen wie *sed* oder *awk*, bei denen die Anzahl der Ausgabedateien eng begrenzt ist, ist dieses Problem nur schwer zu lösen. Da *lex* den Dateinamen aus einem schwierigen Kontext bestimmen kann, ist diese Lösung wesentlich einfacher zu modifizieren als ein handgeschriebenes C-Programm.

2.6 Fehlersuche bei der lexikalischen Analyse

Als Funktion zur lexikalischen Analyse in einem Compiler sollte **yylex()** als Funktion entworfen werden, die ein **int**-Resultat liefert. Bei jedem Aufruf muß das nächste Symbol aus der Eingabe bestimmt, codiert und als Funktionswert geliefert werden. Jedes Muster wird jetzt so entworfen, daß es ein oder mehrere Eingabesymbole erkennt und die zugehörige Aktion enthält dann eine **return**-Anweisung, die einen entsprechenden Funktionswert liefert. Wir müssen dafür sorgen, daß *alle* Eingabezeichen erkannt werden, auch wenn sie weder zu Eingabesymbolen gehören noch stillschweigend ignoriert werden sollen, da **yylex()** in einem Compiler natürlich alle Eingabezeichen verschwinden lassen muß.

Eine derartige Funktion ist nicht leicht zu testen - die Funktionswerte sind Zahlen und als solche im allgemeinen nicht gerade leicht zu merken. Wir fanden hier einen Trick in der C-Programmierung sehr nützlich.[1] Zur Vereinfachung der Fehlersuche wählen wir für das *lex*-Programm folgende prinzipielle Architektur:

```
%{
#include <assert.h>
#define token(x)        (int) "x"

main()
{       char * p;

        assert(sizeof(int) >= sizeof(char *));
        while (p = (char *) yylex())
                printf("%s is \"%s\"\n", p, yytext);
}
%}

%%

Muster          return token(MNEMONIC);
```

Wie das *kleinC*-Beispiel noch zeigen wird, kann man eine Funktion zur lexikalischen Analyse so konditionalisieren, daß sie jederzeit zu Testzwecken in der oben angegebenen Form übersetzt werden kann.

---

[1] Leider funktioniert der Trick auf Maschinen wie der MC 68000 nicht, bei denen Zeigerwerte nicht in **int**-Variable passen. Der Bücherei-Makro **assert()** wird dazu verwendet, diese Situation wenigstens zu erkennen; dieser Makro terminiert das Programm, wenn die angegebene Bedingung nicht zutrifft.

Der Aufrufer von **yylex()** erhält den Wert **MNEMONIC** in druckbarer Form als Resultatwert von **yylex()**. Die einfache **main()**-Funktion zeigt dann den Eingabetext in **yytext[]** zusammen mit der dekodierten Repräsentierung des Resultatwerts zur Fehlersuche.

Nach Konvention muß **yylex()** am Dateiende den Wert **0** liefern. *lex* konstruiert die Funktion **yylex()** so, daß dies intern passiert. **main()** wird hier beendet, wenn **yylex()** dieses Resultat liefert.

Noch ein letzter Hinweis: die reservierten Worte einer Programmiersprache kann man besonders leicht als einzelne Muster schreiben. Leider vergrößert eine lange Liste solcher Muster, die sich selbst darstellen, das von *lex* generierte Programm geradezu dramatisch. Wesentlich effizienter sammelt man reservierte Worte und Namen mit dem gleichen, einzelnen Muster und untersucht die Resultate mit einer kleinen C-Funktion. Eine Standardlösung für dieses Problem befindet sich im nächsten Abschnitt.

2.7 Beispiel

Von der *yacc*-Definition für *kleinC* her wissen wir, welche Eingabesymbole bei der lexikalischen Analyse in diesem Compiler erkannt werden müssen: **yylex()** muß alle Symbole finden, die in **%token**-Anweisungen definiert werden und außerdem noch alle einzelnen Zeichen, die als Eingabesymbole mit Anführungsstrichen direkt angegeben wurden. Wenn wir die im vorigen Abschnitt skizzierte Technik zur Fehlersuche verwenden, müssen wir im Moment die exakten Funktionswerte noch nicht berücksichtigen, die später als Codierung der verschiedenen Eingabesymbole geliefert werden müssen.

Die Muster sind leicht zu formulieren, da wir später ja noch den C-Preprozessor vor unserem Compiler verwenden. Der C-Preprozessor entfernt Kommentare(!) und wertet wie üblich **#define**, **#include** und Anweisungen zur bedingten Übersetzung aus.

Hier folgt nun die komplette Funktion zur lexikalischen Analyse von *kleinC*. **DEBUG** muß definiert werden, wenn wir das Programm zu Testzwecken übersetzen. Einige Zeilen in der nachstehenden Datei sollten deshalb jetzt noch ignoriert werden; wir werden sie im Kapitel 3 erklären, wenn wir dann unseren Parser für *kleinC* aus den *lex*- und *yacc*-Definitionen zusammenfügen.

```
%{
/*
 *      kleinC -- lexikalische Analyse
 */
#ifdef  DEBUG   /* Testversion - falls `assert' es erlaubt */

#       include <assert.h>

        main()
        {       char * p;

                assert(sizeof(int) >= sizeof(char *));

                while (p = (char *) yylex())
                        printf("%-10.10s is \"%s\"\n",
                                p, yytext);
        }
        s_lookup() {}
        yymark() {}
        int yynerrs = 0;

#       define  token(x)        (int) "x"

#else   ! DEBUG         /* Version fuer Compiler */

#       include "y.tab.h"
#       define  token(x)        x

#endif  DEBUG

#define END(v)  (v-1 + sizeof v / sizeof v[0])
static int screen();
%}

letter                          [a-zA-Z_]
digit                           [0-9]
letter_or_digit                 [a-zA-Z_0-9]
white_space                     [ \t\n]
blank                           [ \t]
other                           .
```

```
%%

^"#"{blank}*{digit}+({blank}+.*)?\n     yymark();

">="                                    return token(GE);
"<="                                    return token(LE);
"=="                                    return token(EQ);
"!="                                    return token(NE);
"+="                                    return token(PE);
"-="                                    return token(ME);
"*="                                    return token(TE);
"/="                                    return token(DE);
"%="                                    return token(RE);
"++"                                    return token(PP);
"--"                                    return token(MM);

{letter}{letter_or_digit}*              return screen();

{digit}+                                { s_lookup(token(Constant));
                                          return token(Constant);
                                        }

{white_space}+                          ;

{other}                                 return token(yytext[0]);

%%

/*
 *      Extraktion der reservierten Worte
 */

static struct rwtable {                 /* Tabelle */
        char * rw_name;                 /* Eingabetext */
        int rw_yylex;                   /* yylex() Wert */
        } rwtable[] = {                 /* muss sortiert sein! */
        "break",        token(BREAK),
        "continue",     token(CONTINUE),
        "else",         token(ELSE),
        "if",           token(IF),
        "int",          token(INT),
        "return",       token(RETURN),
        "while",        token(WHILE),
        };
```

```
static int screen()
{       struct rwtable * low = rwtable,
                * high = END(rwtable),
                * mid;
        int c;

        while (low <= high)
        {       mid = low + (high-low)/2;
                if ((c = strcmp(mid->rw_name, yytext)) == 0)
                        return mid->rw_yylex;
                else if (c < 0)
                        low = mid+1;
                else
                        high = mid-1;
        }
        s_lookup(token(Identifier));
        return token(Identifier);
}
```

Befindet sich dieser Text in der Datei *kleinc.l*, dann übersetzt man die Funktion zur lexikalischen Analyse für Testzwecke wie folgt:

```
lex kleinc.l
cc -DDEBUG lex.yy.c -ll -o lexi
```

Wir können einige Testprogramme mit dieser Funktion bearbeiten um zu sehen, daß auch die richtigen Eingabesymbole erkannt werden. Das kleinste komplette Programm ist folgendes:

```
main() {}
```

Die Testversion der lexikalischen Analyse produziert dafür folgende Ausgabe:

```
Identifier is "main"
yytext[0]  is "("
yytext[0]  is ")"
yytext[0]  is "{"
yytext[0]  is "}"
```

Alle Operatoren, die durch einzelne Zeichen repräsentiert werden, erkennen wir sehr leicht: falls kein anderes (früheres) Muster ein Zeichen erkennt, wird es selbst als Funktionswert von **yylex()** geliefert. Dadurch werden natürlich auch alle Zeichen erkannt, die als Eingabezeichen eigentlich nicht erlaubt sind. Wir erhalten so eine kompakte Funktion zur lexikalischen Analyse. Alle Eingabefehler werden wir dann später systematisch behandeln.

Einige andere Aspekte dieses Programms sind vielleicht noch erwähnenswert. *lex* hat rudimentäre Einrichtungen für Textersatz innerhalb der Muster. Eine Zeile der Form

```
Name    Ersatztext
```

im ersten Teil der Eingabe zu *lex* definiert für **Name** einen **Ersatztext**. **Name** kann in Mustern dann als

```
{Name}
```

angegeben werden, damit *lex* statt dessen **Ersatztext** einfügt. Wir benutzen dies sehr oft um unsere Muster klarer zu gestalten.

Das erste Muster in der vorgestellten Funktion befaßt sich mit Zeilen der Form

```
# Zeilennummer Dateiname
```

Solche Zeilen produziert der C-Preprozessor als Positionsangaben, wenn Dateien eingefügt oder Teile der Quelle bei der Übersetzung ausgeschlossen werden. Wie wir im Kapitel 3 noch näher ausführen, ist **yymark()** eine Funktion, die wir implementiert haben um die nötige Information für unsere eigenen Fehlermeldungen aus diesen Positionsangaben zu extrahieren.

Wie versprochen, verwenden wir eine einfache Funktion **screen()**, um mit Hilfe einer sortierten Tabelle reservierte Worte als Sonderfälle aller Eingabesymbole zu erkennen, die das allgemeine Muster für Namen repräsentiert. **screen()** ist als Vorlage zur Lösung derartiger Auswahlprobleme gedacht. Man beachte, daß die **token()**-Technik zur Fehlersuche auch in der Tabelle für reservierte Worte eingesetzt werden kann.

s_lookup() ist ein Anknüpfungspunkt für einen Symboltabellen-Modul; damit werden wir uns im Kapitel 5 befassen.

2.8 Aufgaben

1. Schreiben Sie ein *lex*-Programm, das ein Pascal-Programm liest und davon nur die durch **(* *)** begrenzten Kommentare ausgibt. Alles andere soll ignoriert werden. Hinweis: wie muß das *lex*-Muster für C-Kommentare abgeändert werden, das wir im Abschnitt 2.3 entwickelt haben? ✏

2. Entwerfen Sie ein *lex*-Muster, das eine Fortran-**REAL**-Konstante erkennt. Eine **REAL**-Konstante ist als Ziffernfolge definiert, die wahlweise einen Dezimalpunkt, einen durch **E** abgesetzten Exponenten, oder auch beides zugleich enthält. Als zusätzliche Schikane sollten Sie berücksichtigen, daß in Fortran überall außerhalb von Anführungsstrichen Zwischenraumzeichen erlaubt sind, sogar innerhalb einer Konstanten, in einem Namen oder in einem reservierten Wort. ✏

3. Verwenden Sie das in Aufgabe 2 entwickelte Muster für ein komplettes *lex*-Programm, das ein Fortran-Programm liest und alle **REAL**-Konstanten in **DOUBLE**

PRECISION umwandelt. Das Programm sollte auch alle **REAL**-Vereinbarungen auf **DOUBLE PRECISION** abändern. (Dabei ist ganz günstig, daß das Wort **REAL** in Fortran nur zur Einleitung einer Vereinbarung dient, wenn man einmal annimmt, daß es nicht unvernünftigerweise als Variablenname verwendet wird.) ✏

4. Schreiben Sie ein *lex*-Programm, das ein Programm in Ihrer Lieblingssprache liest, alle reservierten Worte erkennt und sie in Großbuchstaben ausgibt. Alle anderen Großbuchstaben sollten dabei in Kleinbuchstaben umgewandelt werden. Hinweis: verwenden Sie eine Funktion wie **screen()** (siehe Abschnitt 2.7) zur Erkennung der reservierten Worte. ✏

5. Schreiben Sie ein *lex*-Programm, das eine Referenzliste für alle Namen in einem Programm ausgibt, das in Ihrer Lieblingssprache geschrieben ist. Hinweis: wenn Ihr *lex*-Programm einen Namen erkennt, sollten Sie mit einer geeigneten C-Funktion diesen Namen in einer Tabelle von Worten suchen, die für die Referenzliste ignoriert werden können, und in einer Liste von Namen, die schon früher in die Referenzliste aufgenommen wurden. Neue Namen muß man dann zu der zweiten Liste hinzufügen. Jeder Eintrag in dieser Liste ist der Beginn einer Kette von Referenzwerten, in denen die entsprechenden Werte von **yylineno** notiert werden. Wenn **yylex()** das Dateiende erreicht, kann die Referenzliste sortiert und ausgegeben werden. ✏

Kapitel 3: Spracherkennung

In diesem Kapitel fügen wir die bisher betrachteten Teile zusammen: eine Grammatik für *yacc*, Angaben zur lexikalischen Analyse für *lex*, verbunden mit einer rudimentären Symboltabelle, ergeben ein Programm zur Erkennung einer Sprache. Wie das Beispiel zeigen wird, kann ein derartiges Programm zum Beispiel zur Konstruktion eines Formatierers für eine Programmiersprache verwendet werden.

Zunächst müssen wir die Funktionsweise des *Parsers* verstehen, den *yacc* aus einer Grammatik konstruiert, in der Datei *y.tab.c* ablegt und in der Datei *y.output* beschreibt. Damit können wir dann im Abschnitt 3.2 zeigen, wie man einen Parser für *kleinC* tatsächlich zusammenfügt.

Es zeigt sich, daß wir eine **main()**-Funktion schreiben müssen, die den Parser aufruft sowie eine Funktion **yyerror()**, die der Parser aufruft, wenn ein Eingabefehler entdeckt wird. Im Abschnitt 3.3 schweifen wir kurz ab und präsentieren eine komfortable Fassung von **main()**, mit der wir optional auch den C-Preprozessor vor unserem eigenen Compiler aufrufen können. Wir konstruieren dort auch eine allgemein verwendbare **yyerror()**-Funktion, die die Position eines Eingabefehlers beim Parser klar identifiziert.

Leider reflektieren Grammatiken nicht immer gleich die wirklichen Absichten des Sprachdesigners. Im Abschnitt 3.4 beschreiben wir daher, wie man Fehler im Parser findet. Es zeigt sich, daß man Information aus der Datei *y.output* und von einer Option zur Ablaufverfolgung, die *yacc* zur Verfügung stellt, zu einer nützlichen Strategie kombinieren kann, mit der man 'Mißverständnisse' in der Grammatik lokalisieren kann.

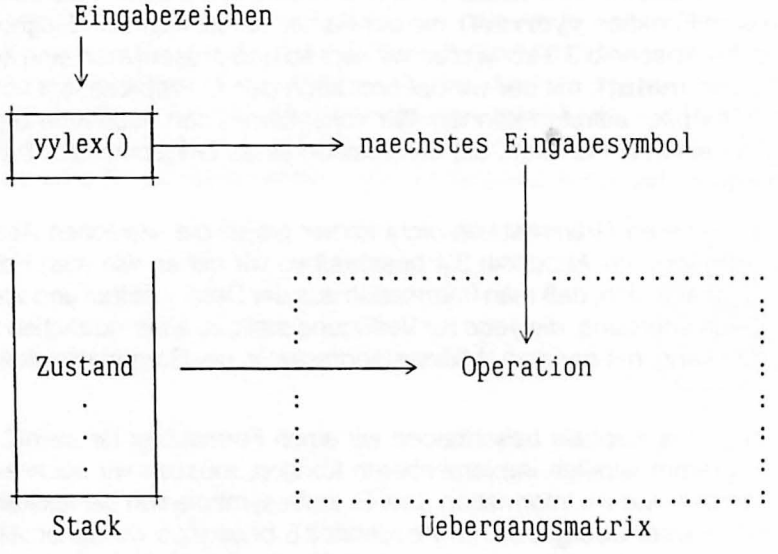
Am Schluß des Kapitels beschreiben wir einen Formatierer für *kleinC*. Bevor wir dieses Programm wirklich implementieren können, müssen wir noch ein Problem lösen, nämlich, wie wir Information *über* Eingabesymbole von der lexikalischen Analyse an den Parser übergeben. Im Abschnitt 3.5 begegnen wir daher *Aktionen* als Eingabe für *yacc* und wir demonstrieren ihre Fähigkeiten gleich, indem wir einen einfachen Tischrechner konstruieren.

3.1 Generierung eines Parsers

Im Abschnitt 1.4 haben wir erklärt, wie *yacc* eine Grammatik analysiert. Die Analyse bestand darin, alle Regeln praktisch gleichzeitig zu verfolgen und dabei in einer Simulation beliebige Eingabesymbole zu akzeptieren. Wenn wir diesem Algorithmus richtige Eingabesymbole vorlegen können, muß die Situation eigentlich einfacher werden: anstatt viele Möglichkeiten gleichzeitig verfolgen zu müssen, kann *yacc* dann einfach in die Zustände übergehen, die durch die Eingabe ausgewählt werden. Das Resultat muß sich zur Erkennung einer Sprache eignen, es ist ein *Parser* oder ein Programm zur *Syntaxanalyse*.

yacc konstruiert im Zuge der Analyse einer Grammatik tatsächlich einen Parser. Der Parser ist ein *push-down*-Automat (eine Stack-Maschine) und besteht aus einem `großen' Stack, auf dem die aktuellen Zustände abgelegt werden, einer Übergangsmatrix, mit der für jede mögliche Kombination von aktuellem Zustand und nächstem Eingabesymbol ein neuer Zustand bestimmt wird, einer Tabelle mit Aktionen, die der Benutzer definieren kann, und die an bestimmten Punkten während der Erkennung ausgeführt werden, und schließlich einem Interpreter, der für die Ausführung dieser Maschine sorgt. Das ganze ist verpackt als Funktion **yyparse()**, die immer wieder eine Funktion **yylex()** zur lexikalischen Analyse aufruft, und die als Funktionswert **0** oder **1** liefert, je nachdem ob als Eingabe ein Satz geliefert wurde oder nicht.

Wir können uns den Parser folgendermaßen vorstellen:



Der aktuelle Zustand, oben auf dem Stack, und das nächste Eingabesymbol, bei Bedarf durch einen Aufruf von **yylex()** erzeugt, wählen eine Operation in der Übergangsmatrix aus. Die Datei *y.output* zeigt den Inhalt der Übergangsmatrix für jedes akzeptable nächste Eingabesymbol und jeden Zustand. In der Übergangsmatrix finden wir fünf Arten von Operationen:

accept

Diese Operation passiert nur einmal, nämlich wenn **$end** das nächste Eingabesymbol ist und wenn wir deshalb den Erkennungsvorgang erfolgreich abschließen. **$end** wird als nicht-positiver Wert von **yylex()** dargestellt.

error

Diese Operation steht überall dort in der Übergangsmatrix, wo ein bestimmtes nächstes Eingabesymbol in einem bestimmten Zustand nicht auftreten darf.

shift *neuer Zustand*

Diese Operation bedeutet, daß das nächste Eingabesymbol im aktuellen Zustand akzeptiert werden kann. Der *neue Zustand* wird auf dem Stack abgelegt und wird zum aktuellen Zustand. Wir sind dadurch in irgendeiner Konfiguration nach rechts vorgerückt.

reduce *Nummer einer Formulierung*

Diese Operation steht bei all den Zuständen in der Übergangsmatrix, die eine komplette Konfiguration enthalten. Die *Nummer einer Formulierung* bezieht sich auf die komplette Konfiguration. In der Datei *y.output* erscheint diese Zahl jeweils im Anschluß an die Formulierung. Die Operation besteht darin, so viele Zustände vom Stack zu entfernen, wie die Formulierung Symbole besitzt. Der dadurch oben auf dem Stack freigelegte Zustand wird neuer aktueller Zustand. Der Grammatikbegriff, dessen Formulierung gerade abgeschlossen wurde, wird vor dem nächsten Eingabesymbol verwendet. Das nächste Eingabesymbol selbst wird erst im Anschluß an den Grammatikbegriff verwendet.

goto *neuer Zustand*

Wie wir gerade sahen, generiert eine **reduce**-Operation implizit einen Grammatikbegriff, der vor dem nächsten Eingabesymbol verwendet werden muß. **goto** ist die **shift**-Operation für diesen Grammatikbegriff. Eine **shift**-Operation verbraucht jeweils das nächste Eingabesymbol. Eine **goto**-Operation benutzt einen Grammatikbegriff und läßt das nächste Eingabesymbol für eine spätere **shift**-Operation unangetastet. Davon abgesehen funktionieren **goto** und **shift** ganz analog: der *neue Zustand* wird auf den Stack gebracht und wird zum aktuellen Zustand.

Um die Operationen im Zusammenhang zu sehen, betrachten wir nochmals die einfache Grammatik, die im Kapitel 1 eingeführt wurde:

```
expression
        : expression '-' IDENTIFIER
        | IDENTIFIER
```

Im Kapitel 1 stellten wir die Datei *y.output* vor, die *yacc* aus dieser Grammatik konstruieren kann. Wir betrachten jetzt nochmals diese Datei um zu sehen, daß sie tatsächlich den Parser vollständig beschreibt. Der Parser beginnt mit **state 0** als aktuellem Zustand auf dem Stack:

```
state 0
        $accept : _expression $end

        IDENTIFIER  shift 2
        .  error

        expression  goto 1
```

Ist das nächste Eingabesymbol ein **IDENTIFIER**, führen wir eine **shift**-Operation aus, das heißt, wir akzeptieren dieses Symbol und bringen **state 2** als aktuellen Zustand auf den Stack. Jedes andere Eingabesymbol würde als **error** betrachtet.

```
state 2
        expression :  IDENTIFIER_    (2)

        .  reduce 2
```

Ohne Rücksicht auf das nächste Eingabesymbol verwenden wir jetzt Formulierung 2 für eine **reduce**-Operation, das heißt, da Formulierung 2 aus einem einzigen Symbol (**IDENTIFIER**) besteht, entfernen wir einen Zustand vom Stack. Nach dem bisherigen Verlauf entdecken wir dadurch **state 0**.

Die **reduce**-Operation hat gerade **expression** als Grammatikbegriff generiert, und die Anweisungen für **state 0** schreiben für diesen Fall eine **goto**-Operation in den **state 1** vor. Wir halten fest, daß das nächste Eingabesymbol, falls es überhaupt noch eines gibt, bisher nicht betrachtet wurde.

```
state 1
        $accept :  expression_$end
        expression :  expression_- IDENTIFIER

        $end  accept
        -  shift 3
        .  error
```

Im **state 1** betrachten wir das Symbol, das dem ersten **IDENTIFIER** in der Eingabe folgt. **$end**, das Ende der Eingabe, oder ein Operator – sind an diesem Punkt vorgesehen. **$end** führt zu einer **accept**-Operation - unser Parser hat **IDENTIFIER** als Satz erkannt! Für den Leser wäre es jetzt sehr instruktiv, mit Hilfe der Datei *y.output* den Aktionen des Parsers bei einem längeren Satz zu folgen.

Ein Problem bleibt: woher weiß **yylex()**, welche Werte **yyparse()** als Repräsentierung der einzelnen Eingabesymbole erwartet? Als natürliche Konvention bietet sich an, einzelne Zeichen als Eingabezeichen durch ihren Wert im Zeichensatz darzustellen, das heißt, die C-Konstante **'x'** soll das Eingabesymbol **'x'** in der Grammatik darstellen, die von *yacc* bearbeitet wird. Für Namen als Eingabesymbole, die durch **%token**-Anweisungen in *yacc* eingeführt werden, müssen sich jedoch **yyparse()** und **yylex()** über die Repräsentierung durch Integer-Werte einigen. Diese Werte müssen von der Darstellung einzelner Zeichen zu unterscheiden sein.

yacc hilft bei der Definition geeigneter Werte. Das Kommando

```
yacc -d grammar.y
```

veranlaßt *yacc*, eine Datei *y.tab.h* zu produzieren, die eine C-Preprozessor **#define**-Anweisung für jeden Namen enthält, der als **%token** eingeführt wurde.[1] Der Ersatztext für einen Namen ist jeweils eine eindeutige Integer-Konstante, größer oder gleich **257**.

Die Datei *y.tab.h* ist in der Datei *y.tab.c* schon enthalten, die *yacc* ebenfalls produziert und die die Funktion **yyparse()** enthält. **yyparse()** und **yylex()** können die gleichen Repräsentierungen für Eingabesymbole verwenden, indem die Datei *y.tab.h* mit einer C-Preprozessor **#include**-Anweisung in die Definition der Funktion zur lexikalischen Analyse eingefügt wird.

Das Verfahren bedingt allerdings, daß Namen, die in der Grammatik für Eingabesymbole verwendet und mit **%token**-Anweisungen eingeführt werden, keine in C reservierten Worte sein können. **yylex()**, die Funktion für die lexikalische Analyse, kann von Hand geschrieben werden, oder sie kann, wie in Kapitel 2 beschrieben wurde, mit Hilfe von *lex* konstruiert werden.

3.2 Beispiel

Im Abschnitt 1.6 haben wir *kleinC* für *yacc* definiert. Nehmen wir an, daß sich diese Definition in der Datei *kleinc.y* befindet. Im Abschnitt 2.7 sahen wir die Datei *kleinc.l*, die eine Funktion zur lexikalischen Analyse von *kleinC* enthält. Wir können jetzt beide Funktionen kombinieren, um einen funktionsfähigen Parser für *kleinC* zu realisieren.

Wenn wir **DEBUG** bei der Übersetzung der Funktion zur lexikalischen Analyse *nicht* definieren, werden die C-Preprozessor-Anweisungen

```
#include "y.tab.h"
#define token(x) x
```

in der Datei *kleinc.l* wirksam. Die erste Anweisung sorgt dafür, daß die von *yacc* in *y.tab.h* abgelegten Definitionen der Eingabesymbole bei der Übersetzung von **yylex()** verwendet werden. Die zweite Anweisung setzt unsere Technik zur Fehlersuche außer Betrieb. Zur Fehlersuche haben wir wie folgt die Namen der Eingabesymbole als Resultatwerte von **yylex()** geliefert:

```
#define token(x) (int) "x"
```

Jetzt liefern wir als Resultat die in *y.tab.h* definierten *Werte* für diese Namen.

Das letzte Muster in *kleinc.l*

```
{other}          return token(yytext[0]);
```

veranlaßt, daß die numerischen Werte all der einzelnen Zeichen übergeben werden, die nicht von einem früheren Muster erfaßt wurden. Das Muster muß ganz am Schluß

[1] Namen als Eingabesymbole können auch dadurch eingeführt werden, daß sie zuerst in **%left**, **%right** oder **%nonassoc**-Anweisungen auftreten. Wir ziehen es aber vor, Eingabesymbole *grundsätzlich* zuerst mit Hilfe von **%token**-Anweisungen zu definieren.

stehen, damit ein einzelner Buchstabe noch als **Identifier** und eine einzelne Ziffer als **Constant** erkannt werden.

Den Parser erzeugen wir etwa mit folgenden Kommandos:

```
lex kleinc.l
yacc -d kleinc.y
cc lex.yy.c y.tab.c -ll
```

Das stimmt allerdings noch nicht ganz. Wenn wir so übersetzen und montieren, holen wir uns die **main()**-Funktion aus der *lex*-Bücherei. Im Kapitel 2 haben wir aber schon gesehen, daß diese Funktion nur **yylex()** einmal und den Parser **yyparse()** überhaupt nicht aufruft! Wir müssen also eine andere **main()**-Funktion verwenden, zum Beispiel

```
main()
{
        yyparse();
}
```

Benutzt man *yacc*, so muß man noch eine weitere Funktion konstruieren. Wenn der Parser eine **error**-Operation in der Übergangsmatrix ausführt, liegt bekanntlich ein Syntaxfehler in der Eingabedatei vor. An dieser Stelle sollte eine Fehlermeldung ausgegeben werden und **yyparse()** enthält dazu folgenden Aufruf:

```
yyerror("syntax error");
```

Es bleibt uns überlassen, eine geeignete Funktion **yyerror()** zu codieren, die die Position des Fehlers in der Eingabe zeigt, usw. Eine minimale Lösung ist etwa:

```
#include <stdio.h>

yyerror(s)
        char * s;
{
        fputs(s, stderr), putc('\n', stderr);
}
```

Die einzelnen Fehler brauchen wir nicht zu zählen. Dies erledigt **yyparse()** selbst in der globalen **int**-Variablen **yynerrs**.

Befinden sich **main()** und **yyerror()** in einer Datei *extra.c*, dann können wir die oben angefangene Übersetzung mit folgendem Kommando vervollständigen:

```
cc extra.c lex.yy.o y.tab.o -ll
```

Der Parser steht in einer Datei *a.out* und kann jetzt ausgeführt werden:

```
a.out
main() { }
^D
```

Nichts passiert, und das ist auch ganz richtig:

```
main() { }
```

ist ein sehr kleines, korrektes *kleinC*-Programm. Unser Parser verhält sich ganz still, wenn als Eingabe wirklich ein Satz angeboten wird. Liegt kein Satz vor, sehen wir die kurze Meldung **syntax error**.

3.3 Hilfsfunktionen

Von der Software-Engineering-Qualität her sind die **main()**- und **yyerror()**-Funktionen aus dem letzten Abschnitt durchaus von *lex* inspiriert. Benutzerfreundliche Compiler sollten wenigstens die Zeilennummer eines Eingabefehlers zeigen und auch als Fehlermeldung sollte ein bißchen mehr als nur **syntax error** ausgegeben werden. In diesem Abschnitt diskutieren wir eine allgemeine Lösung dieses Problems.

Beginnen wir mit der **main()**-Funktion. Es zeigt sich, daß es ganz günstig ist, wenn man immer den C-Preprozessor verwenden kann um Dateien einfügen, Textersatz vornehmen und Teile der Quelle von der Übersetzung ausnehmen zu lassen. Falls die **-C** Option nicht angegeben wird, entfernt der C-Preprozessor auch Kommentare im Stil von C. Unsere Implementierung der **main()**-Funktion ruft den C-Preprozessor auf, wenn wenigstens eine Option angegeben ist, die der C-Preprozessor kennt. Mit Hilfe der Option **-P** oder **-E** können wir also immer den C-Preprozessor vor unserem eigenen Compiler aufrufen:

```
/*
 *      main() -- auf Wunsch C-Preprozessor vor yyparse()
 */

#include <stdio.h>


static usage(name)
        register char * name;
{
        fputs("usage: ", stderr);
        fputs(name, stderr);
        fputs(" [C Preprozessor Optionen] [Datei]\n", stderr);
        exit(1);
}
```

```
main(argc, argv)
        int argc;
        char ** argv;
{       char ** argp;
        int cppflag = 0;

        for (argp = argv; *++argp && **argp == '-'; )
                switch ((*argp)[1]) {
                default:
                        usage(argv[0]);
                case 'C':
                case 'D':
                case 'E':
                case 'I':
                case 'P':
                case 'U':
                        cppflag = 1;
                }
        if (argp[0] && argp[1])
                usage(argv[0]);
        if (*argp && ! freopen(*argp, "r", stdin))
                perror(*argp), exit(1);
        if (cppflag && cpp(argc, argv))
                perror("C Preprozessor"), exit(1);
        exit(yyparse());
}
```

Die Funktion überprüft alle Argumente. Optionen beginnen immer mit –. Ist eine Option angegeben, die der C-Preprozessor kennt, wird der C-Preprozessor mit Hilfe der weiter unten beschriebenen Funktion **cpp()** aufgerufen; diese Funktion erhält dann Zugriff auf alle Argumente. Im Anschluß an alle Optionen kann ein Dateiname angegeben werden; diese Datei würde dann als Eingabedatei verwendet werden. Nach Verarbeitung der Argumente ruft **main()** **yyparse()** auf und liefert den Resultatwert dieser Funktion dann mit **exit()** als Exit-Code des Prozesses.

Zu Testzwecken ist diese **main()**-Funktion sehr gut geeignet. Nähert sich der Compiler seiner Vollendung, wird diese Funktion meist noch erweitert, zum Beispiel mit zusätzlichen Optionsargumenten und mit kurzen Hinweisen zur Benutzung, falls das Programm mit unbekannten Argumenten aufgerufen wird.

Die Funktion **cpp()**, die den C-Preprozessor aufruft, ist etwas komplizierter. Eine elegante Lösung, die temporäre Dateien vermeidet, hängt davon ab, daß man eine Pipeline so anschließen kann, daß die Funktion **yylex()** implizit gezwungen wird, aus dieser Pipeline, also vom C-Preprozessor, zu lesen. Nach Definition liest **yylex()** einzelne Zeichen durch Aufruf einer Funktion **input()**, die als Makro von *lex* definiert wird. Es zeigt sich, daß diese Funktion ihrerseits den globalen Filepointer **yyin** verwendet, der **extern** zugänglich ist. Dies kann man dann in **cpp()** ausnützen:

```
/*
 *      cpp() -- lex-Eingabe mit C-Preprozessor bearbeiten
 */

#include <stdio.h>

#ifndef CPP             /* Dateiname fuer C-Preprozessor */
#       define  CPP     "/lib/cpp"
#endif

int cpp(argc, argv)
        int argc;
        char ** argv;
{       char ** argp, * cmd;
        extern FILE * yyin;     /* input() in lex */
        extern FILE * popen();
        int i;

        for (i = 0, argp = argv; *++argp; )
                if (**argp == '-' &&
                    index("CDEIUP", (*argp)[1]))
                        i += strlen(*argp) + 1;

        if (! (cmd = (char *)
                        calloc(i + sizeof CPP, sizeof(char))))
                return -1;      /* kein Platz */

        strcpy(cmd, CPP);
        for (argp = argv; *++argp; )
                if (**argp == '-' &&
                    index("CDEIPU", (*argp)[1]))
                        strcat(cmd, " "), strcat(cmd, *argp);

        if (yyin = popen(cmd, "r"))
                i = 0;          /* alles in Ordnung */
        else
                i = -1;         /* kein Preprozessor */
        cfree(cmd);
        return i;
}
```

cpp() mißt zunächst die Länge aller Optionen, die der C-Preprozessor kennt. Anschließend beschafft sich die Funktion Speicherplatz, verkettet die Optionen und konstruiert ein Kommando zum Aufruf des C-Preprozessors mit diesen Optionen. Mit der Büchereifunktion **popen()** wird der C-Preprozessor als Filter mit **yyin** ver-

bunden, die Zeichenkette mit dem Kommando wird wieder freigegeben und **cpp()** liefert **0** als Funktionswert, wenn dies alles funktioniert hat.

Wenden wir uns nun den Fehlermeldungen zu. Als erstes zeigen wir, wie man am Beginn jeder Meldung eine vernünftige Positionsangabe erzeugen kann. Dabei sollten wir den Namen der Eingabedatei ausgeben, insbesondere wenn Dateien mit dem C-Preprozessor bearbeitet werden. Die Nummer der aktuellen Eingabezeile, **yylineno**, wird von **yylex()** korrekt festgehalten, wenigstens solange kein Preprozessor im Spiel ist. Das aktuelle oder das nächste Eingabesymbol befindet sich in **yytext[]**, aber dieses Symbol kann auch ein Zeilentrenner sein; am Dateiende kann dieser Vektor sogar leer sein. In beiden Fällen ist die tatsächliche Zeilennummer des Fehlers **yylineno**. Man kann viel Information praktisch automatisch generieren, nur muß man das ganze sehr sorgfältig formatieren:

```
/*
 *      yywhere() -- Eingabeposition fuer yyparse()
 *      yymark() -- `# Zeile Datei' speichern
 */

#include <stdio.h>

extern FILE * yyerfp;           /* Fehlermeldungen */

extern char yytext[];           /* aktuelles Symbol */
extern int yyleng;              /* und Laenge */
extern int yylineno;            /* aktuelle Eingabezeile */

static char * source;           /* aktuelle Eingabedatei */

yywhere()                       /* Positionsmeldung */
{       char colon = 0;         /* Trenner notwendig ? */

        if (source && *source && strcmp(source, "\"\""))
        {       char * cp = source;
                int len = strlen(source);

                if (*cp == '"')
                        ++cp, len -= 2;
                if (strncmp(cp, "./", 2) == 0)
                        cp += 2, len -= 2;
                fprintf(yyerfp, "Datei %.*s", len, cp);
                colon = 1;
        }
```

```
        if (yylineno > 0)
        {       if (colon)
                        fputs(", ", yyerfp);
                fprintf(yyerfp, "Zeile %d",
                        yylineno -
                        (*yytext == '\n' || ! *yytext));
                colon = 1;
        }
        if (*yytext)
        {       register int i;

                for (i = 0; i < 20; ++ i)
                        if (!yytext[i] || yytext[i] == '\n')
                                break;
                if (i)
                {       if (colon)
                                putc(' ', yyerfp);
                        fprintf(yyerfp, "bei \"%.*s\"",
                                i, yytext);
                        colon = 1;
                }
        }
        if (colon)
                fputs(": ", yyerfp);
}

yymark()                /* aus `# Zeile Datei' extrahieren */
{
        if (source)
                cfree(source);
        source = (char *) calloc(yyleng+1, sizeof(char));
        if (source)
                sscanf(yytext, "# %d %s", &yylineno, source);
}
```

Wir definieren **yyerfp** als separaten Filepointer, der für alle Meldungen benutzt wird. Beim Entwurf des Compilers kann man dann immer noch die Meldungen auf Standard-Ausgabe (wie hier voreingestellt) lenken, oder man kann sie in einer eigenen Datei ausgeben, indem man einfach einen anderen Filepointer an **yyerfp** zuweist.

yymark() ist eine Funktion, die den Namen der Eingabedatei und die Zeilennummer den Positionsstempeln entnimmt, die der C-Preprozessor erzeugt (siehe Abschnitt 2.7).

Wenn alle möglichen Bestandteile wirklich ausgegeben werden, hat die Positionsangabe, die **yywhere()** produziert, folgende Form:

```
Datei source.c, Zeile 10 bei "badsymbol":
```

Dabei ist **badsymbol** in der Regel das Eingabesymbol, das dem Fehler folgt.

Wir können jetzt schließlich eine allgemeine **yyerror()**-Funktion implementieren, die die Position eines Fehlers etwas besser bezeichnet:

```
#include <stdio.h>

FILE * yyerfp = stdout;          /* Fehlermeldungen */

yyerror(s)
        register char * s;
{       extern int yynerrs;      /* gesamte Fehlerzahl */

        fprintf(yyerfp, "[Fehler %d] ", yynerrs+1);
        yywhere();
        fputs(s, yyerfp);
        putc('\n', yyerfp);
}
```

yynerrs ist ein Zähler, den **yyparse()** manipuliert; dieser Zähler wird im Anschluß an jeden Aufruf von **yyerror()** inkrementiert. Gerade im Protokoll einer längeren Übersetzung erweist es sich als sehr hilfreich, wenn die Fehlermeldungen durchnumeriert sind.

3.4 Fehlersuche im Parser

Der Parser kann jetzt ausprobiert werden. Leider passiert schlicht gar nichts, wenn wir unserem Parser eine korrekte Eingabedatei zur Bearbeitung anbieten. Ist die Eingabedatei fehlerhaft, erhalten wir eine einzige, mehr oder weniger nützliche Fehlermeldung. Wenn wir dann glauben, daß die Eingabedatei in Wirklichkeit korrekt war, während eine Fehlermeldung trotzdem ausgegeben wird, werden die Dinge problematisch: wir müssen herausfinden, warum **yyparse()** beschloß, die Fehlermeldung zu erzeugen.

Mit der im Abschnitt 2.6 beschriebenen Technik können wir nachprüfen, daß die lexikalische Analyse keine Schuld trifft, das heißt, wir können nachweisen, daß **yyparse()** wirklich die Symbole erhält, die unserer Meinung nach in der Eingabedatei sind. Funktioniert die lexikalische Analyse wie gewünscht, können wir eine Hilfe zur Fehlersuche benützen, die *yacc* zur Verfügung stellt: bei der Übersetzung des Parsers, also der Datei *y.tab.c*, definieren wir dazu das Symbol **YYDEBUG**:

```
cc -DYYDEBUG y.tab.c lex.yy.c main.c yyerror.c -ll
```

Die resultierende **yyparse()**-Funktion enthält dann Anweisungen zur Ablaufverfolgung, die wirksam werden, wenn man an die **int**-Variable **yydebug** einen von Null verschiedenen Wert zuweist. **yydebug** ist eine globale Variable und kann zum Beispiel mit *adb* verändert werden:

```
adb -w a.out
yydebug?w 1
$q
```

Wir illustrieren die Resultate der Ablaufverfolgung mit leicht modifizierten Versionen der im Abschnitt 1.4 eingeführten Grammatik für arithmetische Ausdrücke. Hier ist eine rudimentäre Funktion zur lexikalischen Analyse:

```
%{
#include "y.tab.h"
%}
%%

[a-z]+          return IDENTIFIER;
[ \t\n]+        ;
.               return yytext[0];
```

Die erste Version der Grammatik definiert – mit **%left** als links-assoziativen Operator:

```
%token  IDENTIFIER
%left   '-'
%%

expression
        : expression '-' expression
        | IDENTIFIER

%%

main()
{
#ifdef  YYDEBUG
        extern int yydebug;

        yydebug = 1;
#endif
        printf("yyparse() == %d\n", yyparse());
}
```

Wie die Eingabe zu *lex* kann auch die Eingabe zu *yacc* optional einen dritten Teil enthalten, dem dann ebenfalls **%%** vorausgehen muß. Dieser Teil der Eingabe wird von *yacc* in *y.tab.c* übertragen. Hier enthält dieser Teil eine **main()**-Funktion, die

yydebug aktiviert, wenn diese Variable existiert. Zur Illustration wird in **main()** auch der Resultatwert von **yyparse()** ausgegeben.

yacc produziert folgende *y.output*-Datei für diesen Parser:

```
state 0
        $accept : _expression $end

        IDENTIFIER  shift 2
        .  error

        expression  goto 1

state 1
        $accept :  expression_$end
        expression :  expression_- expression

        $end  accept
        -  shift 3
        .  error

state 2
        expression :  IDENTIFIER_    (2)

        .  reduce 2

state 3
        expression :  expression -_expression

        IDENTIFIER  shift 2
        .  error

        expression  goto 4

state 4
        expression :  expression_- expression
        expression :  expression - expression_    (1)

        .  reduce 1
```

Die statistische Information wurde hier unterdrückt.

Nehmen wir an, daß die Eingaben für *lex* und *yacc* in den Dateien *exp.l* und *exp.y* stehen. Wir konstruieren den Parser und bearbeiten damit eine korrekte Eingabe:

```
yacc -dv exp.y
lex exp.l
cc -DYYDEBUG y.tab.c lex.yy.c yyerror.c -ll -o exp
exp
a - b - c
^D
```

Zusammen mit *y.output* zeigt die Ablaufverfolgung[1] sehr genau, wie die Eingabe verarbeitet wird:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 3
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 3
[yydebug] push state 4
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 3
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 3
[yydebug] push state 4
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
[yydebug] reading [end of file]
yyparse() == 0
```

[1] Die Abschnitte 8 und 9 im Anhang beschreiben leichte Modifikationen für */usr/lib/yaccpar*, die wir durchgeführt haben, um die vom Parser produzierten Fehlermeldungen und die Ausgabe bei der Ablaufverfolgung zu verbessern. Im ganzen Buch zeigen wir die Ausgabe, die diese verbesserte Version produziert. Die Standard-Version zeigt die numerischen Werte der Eingabesymbole und in manchen Fällen wird die Eingabe bereits zerstört *bevor* sie zur Ablaufverfolgung ausgegeben wird! (Um die Beispiele realistisch zu halten, haben wir die Ausgabe der Ablaufverfolgung nicht ins Deutsche übertragen. A.d.Ü.)

Man sieht leicht, daß im **state 4** die Formulierung **(1)**

```
expression : expression '-' expression
```

reduziert wird *bevor* der zweite – Operator eingelesen wird. Die Erkennung verläuft erfolgreich und **yyparse()** liefert Null als Funktionswert.

Betrachten wir als Gegensatz die gleiche Grammatik, bei der aber – mit **%right** rechts-assoziativ vereinbart wird:

```
%token  IDENTIFIER
%right  '-'
%%

expression
        : expression '-' expression
        | IDENTIFIER
```

Mit Ausnahme von **state 4** bleibt die Übergangsmatrix unverändert:

```
state 4
        expression :  expression_- expression
        expression :  expression - expression_    (1)

        -  shift 3
        .  reduce 1
```

Jetzt ist – rechts-assoziativ. Die am weitesten rechts gelegene Phrase mit – muß zuerst reduziert werden, folglich müssen zuerst *alle* – Operatoren auf dem *yacc*-Stack gesammelt werden. Der in **yyparse()** verwendete Algorithmus benötigt für links-rekursive Formulierungen und links-assoziative Operatoren weniger Platz auf dem Stack.

Die gleiche Eingabe wie oben liefert jetzt folgende Ablaufverfolgung:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 3
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 3
[yydebug] push state 4
[yydebug] reading '-'
[yydebug] push state 3
[yydebug] reading IDENTIFIER
```

```
[yydebug] push state 2
[yydebug] reduce by (2), uncover 3
[yydebug] push state 4
[yydebug] reading [end of file]
[yydebug] reduce by (1), uncover 3
[yydebug] push state 4
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
yyparse() == 0
```

Man sieht leicht, daß nun die beiden interessanten Reduzierungen mit Formulierung **(1)** erst stattfinden *nachdem* das Dateiende bereits erreicht wurde.

Die Ablaufverfolgung ist vielleicht lehrreicher, wenn wir eine fehlerhafte Eingabe verwenden:

```
a - -
```

Im links-assoziativen Fall erhalten wir

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 3
[yydebug] reading '-'
[Fehler 1] Zeile 1 bei "-": erwartet wird: IDENTIFIER
[yydebug] recovery pops 3, uncovers 1
[yydebug] recovery pops 1, uncovers 0
[yydebug] recovery pops 0, stack is empty
yyparse() == 1
```

Wir betrachten *y.output* und sehen, daß in **state 3** nur ein **IDENTIFIER** vom Parser akzeptiert werden kann - dies besagt auch unsere verbesserte Fehlermeldung. Die Ablaufverfolgung zeigt, wie wir in **state 3** geraten sind, und sie zeigt auch, daß **yyparse()** den ganzen Stack eliminiert, wenn ein Eingabefehler gefunden wird. Der Funktionswert ist in diesem Fall **1**. Mit der Fehlerbehandlung werden wir uns noch genauer im Kapitel 4 befassen.

Zum Schluß betrachten wir noch folgende Modifikation unserer Grammatik:

```
%token  IDENTIFIER
%nonassoc '-'
%%
expression
        : expression '-' expression
        | IDENTIFIER
```

Jetzt wurde – mit **%nonassoc** als nicht-assoziativ markiert, das heißt, jetzt soll eine Folge von Subtraktionen nicht erlaubt sein. *y.output* zeigt eine entsprechende Änderung im Zustand 4 in der Übergangsmatrix:

```
state 4
        expression :  expression_- expression
        expression :  expression - expression_    (1)

        -  error
        .  reduce 1
```

Wenn wir die Bearbeitung von

```
a - b - c
```

verfolgen, sehen wir, wo diese Grammatik unlogisch ist:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 3
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 3
[yydebug] push state 4
[yydebug] reading '-'
[Fehler 1] Zeile 1 bei "-": syntax error
[yydebug] recovery pops 4, uncovers 3
[yydebug] recovery pops 3, uncovers 1
[yydebug] recovery pops 1, uncovers 0
[yydebug] recovery pops 0, stack is empty
yyparse() == 1
```

Mit **a** erfolgt eine **shift**-Operation in **state 2**, wo dieser **IDENTIFIER** als **expression** reduziert wird. Zurück in **state 0** akzeptieren wir die **expression** und eine **goto**-Operation führt uns in **state 1**. – **b** kann akzeptiert werden, wir kommen mit **shift**-Operationen aus **state 1** in **state 3** und **state 2**, wo der **IDENTIFIER b** wieder als **expression** reduziert wird. Diesmal entdecken wir auf dem Stack aber **state 3** und **expression** führt jetzt mit einer **goto**-Operation in **state 4**. Hier wählt jetzt – die **error**-Operation in der Übergangsmatrix!

Würden wir jetzt nach einem Defekt in der Grammatik suchen, müßten wir uns überlegen, *warum* – in **state 4** nicht akzeptiert werden kann. Wenn wir **state 1** und **state 4** vergleichen, sehen wir daß die Konfiguration

```
expression : expression_- expression
```

für – zu zwei verschiedenen Einträgen in der Übergangsmatrix führt. Dies ist nur möglich, wenn eine seltsame Vorschrift gegen Mehrdeutigkeit angegeben wurde: damit haben wir aber unseren 'Fehler' entdeckt.

Das Beispiel ist vielleicht obskur, aber die Technik zur Fehlersuche ganz und gar nicht. Die Praxis zeigt, daß man normalerweise mißverstandene Stellen in der Grammatik sehr leicht findet, wenn man die durch **yydebug** aktivierte Ablaufverfolgung mit *y.output* vergleicht. Das letzte Beispiel, nämlich den Effekt merkwürdiger Vorschriften gegen Mehrdeutigkeit zu verstehen, ist wahrscheinlich schon die schwierigste Aufgabe für eine Fehlersuche mit Hilfe der Ablaufverfolgung.

3.5 Bezeichner

Identifier und **Constant** sind in Wirklichkeit Klassen von Eingabesymbolen. Den Parser interessiert nur die Klasse als Ganzes, um damit einen Satz erkennen zu können, aber wir brauchen später das tatsächlich angegebene Eingabesymbol zur weiteren Bearbeitung. Entwickeln wir den obligaten, einfachen Tischrechner ohne Variablen:

```
/*
 *      Tischrechner
 */

%token  Constant
%left   '+' '-'
%left   '*' '/'
%%

line
        : /* leer */
        | line expression '\n'
                { printf("%d\n", $2); }

expression
        : expression '+' expression
                { $$ = $1 + $3; }
        | expression '-' expression
                { $$ = $1 - $3; }
        | expression '*' expression
                { $$ = $1 * $3; }
        | expression '/' expression
                { $$ = $1 / $3; }
        | '(' expression ')'
                { $$ = $2; }
        | Constant
                /* $$ = $1; */
```

Der in geschweifte Klammern eingeschlossene Text könnte als Kommentar angesehen werden, der beschreibt, wie eine Formulierung interpretiert werden soll:

```
expression : expression '+' expression
{   $$     =     $1      +      $3;  }
```

'Finden wir mit + verknüpfte **expressions**, müssen wir deren Werte addieren; die Summe ist der Wert der reduzierten **expression**.'

```
expression : '(' expression ')'
{   $$     =         $2;       }
```

'Finden wir **expression** in Klammern eingeschlossen, ist der Wert der reduzierten **expression** der des *zweiten* Symbols in der Formulierung.'

```
expression : Constant
/*  $$     =   $1; */
```

'Finden wir **Constant**, ist der Wert der reduzierten **expression** der des *ersten* Symbols in der Formulierung, also der Wert der **Constant**.'

```
line : line expression '\n'
{ printf("%d\n", $2     );}
```

'Finden wir eine vollständige Instanz von **line**, geben wir den Wert des *zweiten* Symbols in der Formulierung aus.'

Tatsächlich stehen in den geschweiften Klammern nicht nur Kommentare. Nach jeder Formulierung läßt uns *yacc* eine *Aktion* angeben, das heißt, C-Anweisungen, die in geschweifte Klammern eingeschlossen sind. Die Aktion wird genau dann ausgeführt, wenn die Formulierung reduziert wird.

yyparse() unterhält einen *Wert-Stack* parallel zum Stack der Zustände, der im Abschnitt 3.1 vorgestellt wurde. Wenn ein Symbol akzeptiert wird, das heißt, wenn der aktuelle Zustand im Zuge einer **shift**- oder **goto**-Operation auf den Stack gebracht wird, wird auch ein Wert auf dem Wert-Stack abgelegt.

Der Wert, der dabei bei einer **shift**-Operation verwendet wird, also für ein Eingabesymbol, wird der globalen **int**-Variablen **yylval** entnommen, die *yacc* definiert; dieser Wert kann bei der lexikalischen Analyse für das Eingabesymbol in **yylval** hinterlegt werden. Für unser Beispiel müssen wir dort den Wert einer **Constant** hinterlegen:

```
%{
/*
 *      lexikalische Analyse fuer Tischrechner
 */
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+          { yylval = atoi(yytext); return Constant; }
[ \t]+          ;
\n              |
.               return yytext[0];
```

Die Büchereifunktion **atoi()** berechnet den Zahlenwert einer Folge von Ziffern. Dieser Wert wird hier an **yylval** als Wert der **Constant** zugewiesen.

Der Wert, der bei einer **goto**-Operation verwendet wird, also für einen Grammatikbegriff, der durch eine **reduce**-Operation generiert wurde, stammt aus der globalen Variable **yyval**, die *yacc* ebenfalls definiert; dieser Wert kann in der Aktion berechnet werden, die im Zuge der **reduce**-Operation ausgeführt wird.

Wie das Beispiel zeigt, benötigt die Aktion dazu normalerweise einige Werte, die auf dem Wert-Stack für Symbole hinterlegt wurden, die in der aktuellen Formulierung gerade reduziert werden. Die Schreibweise **$***i* in einer Aktion bezeichnet den Wert vom Wert-Stack, der für das Symbol in Position *i* in der Formulierung hinterlegt wurde. **$$** bezeichnet **yyval**, also den Wert, der auf dem Wert-Stack hinterlegt werden wird, wenn der neue Grammatikbegriff mit einer **goto**-Operation im Anschluß an die Reduzierung akzeptiert wird.

Die Aktion

```
{ $$ = $1; }
```

wird als Voreinstellung durchgeführt. Diese Aktion besagt, daß der Wert des ersten Symbols in der Formulierung für den Grammatikbegriff auf den Wert-Stack gebracht wird, in den die Formulierung reduziert wird.

Unser Tischrechner funktioniert wie vorgesehen. Für jede **Constant** wird der Zahlenwert bei der lexikalischen Analyse an **yylval** zugewiesen; von dort legt ihn **yyparse()** auf dem Wert-Stack ab. Wird eine Formulierung wie zum Beispiel

```
expression : expression '-' expression
```

reduziert, berechnet die zugehörige Aktion

```
{ $$ = $1 - $3; }
```

die entsprechende Differenz, die dann im Anschluß an die Reduzierung auf den Wert-Stack gebracht wird. Der Aufruf von **printf()** am Schluß zeigt den Wert eines arithmetischen Ausdrucks, der auf einer Zeile dem Tischrechner eingegeben wurde.

3.6 Typdefinition für den Wert-Stack

yylval, **yyval** und der Wert-Stack können sehr verschiedenartige Information beinhalten. Als Voreinstellung besteht der Wert-Stack aus **int**-Elementen. Für unseren Tischrechner wären wohl **double**-Elemente interessanter. In einem Compiler enthält **yylval** wahrscheinlich einen Zeiger auf einen Symboltabelleneintrag für jedes 'längliche' Eingabesymbol. Leider können **double**-Werte - und auf manchen Maschinen sogar Zeigerwerte - nicht in **int**-Variablen abgelegt werden.

Der Wert-Stack, den der Parser realisiert, kann in einer Eingabe für *yacc* mit beliebigem Typ definiert werden. [1] In diesem Abschnitt beschreiben wir eine Methode, die den Wert-Stack für *yacc* völlig unsichtbar deklariert. Ein wesentlich aufwendigeres Verfahren zur Deklaration des Typs, bei dem dann *yacc* ausgedehnte semantische Prüfungen vornimmt, wird im Abschnitt 5.4 beschrieben.

yylval, **yyval** und der Wert-Stack werden im Parser mit dem Typ **YYSTYPE** vereinbart. **YYSTYPE** selbst wird *mit dem C-Preprozessor* als **int** vereinbart, falls im ersten Teil der Eingabe für *yacc* dieser Name nicht schon definiert wurde. Für *yacc* unsichtbar kann man diese Definition folgendermaßen liefern:

```
%{
#define YYSTYPE double
%}

%%

/* Grammatik, usw. */
```

Genau wie *lex* kopiert auch *yacc* im ersten Teil seiner Eingabe Text unverändert, der in **%{** und **%}** eingeschlossen ist. Der Text wird im Parser ziemlich am Anfang eingefügt, so daß voreingestellte Definitionen in *yacc* entsprechend korrigiert werden können.

Wenn wir den Wert-Stack als **double** vereinbaren, müssen wir natürlich die lexikalische Analyse geringfügig ändern. **yylval** ist jetzt auch **double** und wir müssen Ziffernketten entsprechend anders umwandeln:

```
%{
/*
 *      lexikalische Analyse fuer `double' Tischrechner
 */

#include "y.tab.h"
extern double yylval, atof();
%}
```

[1] ...vorausgesetzt der zugehörige C-Compiler erlaubt die Zuweisung von Strukturen! A.d.Ü.

```
digits          ([0-9]+)
pt              "."
sign            [+-]?
exponent        ([eE]{sign}{digits})

%%

{digits}({pt}{digits}?)?{exponent}?     |
{digits}?{pt}{digits}{exponent}?        { yylval = atof(yytext);
                                          return Constant;
                                        }

[ \t]+          ;
\n              |
.               return yytext[0];
```

Das Beispiel zeigt, wie man eine ziemlich allgemeine Klasse von Gleitkommakonstanten mit *lex* erkennt und mit der Büchereifunktion **atof()** umwandelt. Ein Gleitkommawert besteht hier aus Ziffern, einem Dezimalpunkt, weiteren Ziffern, und einem optionalen Exponenten, der optional auch ein Vorzeichen besitzen kann.

Einen Typ für den Wert-Stack auf diese Weise zu vereinbaren, ist sehr bequem und bedingt keine Änderungen der Grammatik selbst. Diese Technik enthält jedoch eine subtile Falle: **YYSTYPE** ist ein mit dem C-Preprozessor definierter Name. Der für den Wert-Stack erwünschte Typ muß als Typname angegeben werden, das heißt, als einfacher Name oder als Strukturvereinbarung. Der Ersatztext für **YYSTYPE** kann insbesondere *keine* Vereinbarung für einen Zeiger sein! Eine derartige Vereinbarung muß man zuerst mit **typedef** mit einem Namen verknüpfen. Erst dieser Name kann dann als Ersatztext von **YYSTYPE** dienen:

```
typedef char *  CHAR_PTR;
#define YYSTYPE CHAR_PTR
```

Hier wird **CHAR_PTR** als Typname verwendet. Dieses Vorgehen ist nicht gerade elegant, aber es ist unabdingbar, da die Definition

```
#define YYSTYPE char *
```

innerhalb von *yaccpar* nicht 'korrekt' wirken würde.

3.7 Beispiel

Wir können jetzt endlich einen *kleinC*-Formatierer entwickeln. Es zeigt sich, daß wir unsere Grammatik leicht ändern müssen: für jedes Eingabesymbol brauchen wir eine Aktion, die das Symbol ausgibt. Dies könnte natürlich während der lexikalischen Analyse viel leichter erledigt werden, aber dann könnten wir manche Anweisungen nicht in Abhängigkeit von Kontext und Syntax einrücken. Ausgaben von

lexikalischer Analyse und Aktionen in der Grammatik zu mischen geht sehr schief: der Parser darf die lexikalische Analyse aufrufen um ein Eingabesymbol voraus zu schauen, und das macht er 'manchmal'.

Als Beispiel, was wir erreichen wollen, betrachten wir folgendes *kleinC*-Programm, das nichts tut, dieses allerdings syntaktisch korrekt und typographisch einigermaßen verwirrend:

```
/*
 *      Formatierbeispiel
 */
main(a,b) int a,b; {
a=b;
{a;b;}
if (a==b) {a;b;}
if (a==b+1) a; else b;
while (a==b) {a; break; continue;}
return;
;
}
int f() {int x;int y; return nothing; }
```

Über Stil beim Einrücken kann man sich bekanntlich streiten. Wir möchten jedenfalls folgende Ausgabe sehen:

```
main(a, b)
        int     a, b;
{
        a = b;
        {
                a;
                b;
        }
        if (a == b)
        {
                a;
                b;
        }
        if (a == b + 1)
                a;
        else
                b;
```

```
        while (a == b)
        {
                a;
                break;
                continue;
        }
        return;
        ;
}

int     f()
{       int     x;
        int     y;

        return nothing;
}
```

Einige Eingabesymbole, insbesondere die geschweiften Klammern und **else**, müssen gesondert behandelt werden, da sie den linken Rand beeinflussen. Hier haben wir ein gutes Beispiel dafür, daß man gelegentlich Kontext-Information zwischen den Aktionen einer Grammatik austauschen muß. **else** rückt den linken Rand ein, aber unser Stil beim Einrücken verlangt, daß die geschweiften Klammern für einen unmittelbar folgenden Block noch auf dem alten linken Rand angeordnet werden sollen. Dadurch sparen wir insgesamt an Papierbreite, aber wir haben doch die hohe Signalwirkung der linken Klammern am linken Rand. Wir reichen etwas Information mit Hilfe von **static** definierten Variablen weiter.

Die Formatieraktionen verwenden bestimmte Worte als Parameter. Diese werden in einer separaten Definitionsdatei *fmt.h* vereinbart, in der auch der Typ für den *yacc*-Wert-Stack vereinbart wird:

```
/*
 *      Parameter fuer Formatiereraufrufe
 */

#define IN      1       /* Rand nach rechts verlegen */
#define EX      (-1)    /* Rand nach links verlegen */
#define AT      0       /* Rand unveraendert */

/*
 *      Typ fuer den yacc-Wert-Stack (Texte!)
 */

typedef char * CHAR_PTR;
#define YYSTYPE CHAR_PTR
```

Der Formatierer beruht auf gewissen Hilfsfunktionen, mit denen die gesamte Ausgabe erzeugt wird. Die **out()**-Funktion könnte außerdem die Breite der generierten Ausgabe überwachen und entsprechend zusätzliche Zeilentrenner oder Seitenvorschübe einfügen. Alle Hilfsfunktionen befinden sich in der Datei *fmt.c*:

```
/*
 *      kleinC - Hilfsfunktionen fuer Formatierer
 */

#include <stdio.h>
#include "fmt.h"

/*
 *      rudimentaere Symboltabelle:
 *              fuer jedes Symbol Text aufbewahren.
 */

s_lookup(yylex)
        int yylex;              /* Constant oder Identifier */
{       extern YYSTYPE yylval;  /* Wert fuer Parser */
        extern char yytext[];   /* Eingabesymbol */
        extern char * strsave();

        yylval = strsave(yytext);
}

/*
 *      Formatieraufrufe:
 *
 *      at(AT)          (ohne Wirkung)
 *      at(IN)          Rand nach rechts verlegen
 *      at(EX)          Rand nach links verlegen
 *      nl(delta)       Zeilentrenner, at(delta)
 *      cond(IN)        nl(IN)
 *      cond(EX)        wenn unmittelbar nach cond(IN): at(EX)
 *      uncond(AT)      nl(AT)
 *      uncond(EX)      nur nicht direkt nach uncond(AT): at(EX)
 *      out(s)          Text s ausgeben
 *
 *      Randpositionierung wirkt unmittelbar vor dem ersten
 *      out(s) auf der neuen Zeile. `cond' Aufrufe sind fuer
 *      geschweifte Klammern und `else'.
 */
```

```
static int lmargin = 0;  /* linker Rand, in \t */
static int atmargin = 1; /* sind wir am linken Rand?? */
static int condflag = 0; /* siehe cond(); = 0 in out() */
static int uncdflag = 0; /* siehe uncond(); = 0 in out() */


at(delta)
        int delta;
{
        lmargin += delta;
}


cond(delta)
        int delta;
{
        switch (delta) {
        case IN:
                ++condflag;
                nl(IN);
                break;
        case EX:
                if (condflag)
                {       at(EX);
                        condflag = 0;
                }
        }
}

uncond(delta)
        int delta;
{
        switch (delta) {
        case AT:
                ++uncdflag;
                nl(AT);
                break;
        case EX:
                if (! uncdflag)
                        at(EX);
        }
}
```

```
nl(delta)
        int delta;
{
        at(delta);
        putchar('\n');
        atmargin = 1;
}

out(s)
        char *s;
{
        if (atmargin)
        {       register int i;

                for (i = 0; i < lmargin; i++)
                        putchar('\t');
                atmargin = 0;
        }
        fputs(s, stdout);
        condflag = uncdflag = 0;
}
```

strsave() ist eine nützliche Funktion [Ker78b], die eine als Argument übergebene Zeichenkette dynamisch speichert. Sie ist im Abschnitt 3 im Anhang angegeben, da sie ihren Weg in die C-Bücherei nicht gefunden hat. Dies genügt als Symboltabelle für unser Problem.

Wir benötigen explizite Regeln für alle Eingabesymbole, mit Aktionen, die jedes Eingabesymbol auch wieder ausgeben. Davon abgesehen bleibt die Grammatik einigermaßen unverändert. Wir zeigen daher nur die Teile, die abgeändert oder hinzugefügt wurden:

```
/*
 *      kleinC
 *      Syntaxanalyse
 *      Formatierung
 *      (s/r Konflikt: bei ELSE)
 */

%{
#include "fmt.h"        /* Formatieraktionen */
%}
```

```
/*
 *      Eingabesymbole
 */

/*
 *      Vorrangtabelle
 */

%%

program

definitions

definition
        : function_definition
        | int function_definition
        | declaration

function_definition
        : identifier lp optional_parameter_list rp
                { nl(IN); }
          parameter_declarations
                { at(EX); }
          compound_statement
                { nl(AT); }

optional_parameter_list

parameter_list
        : identifier
        | parameter_list co identifier

parameter_declarations

parameter_declaration
        : int parameter_declarator_list sc

parameter_declarator_list
        : identifier
        | parameter_declarator_list co identifier
```

```
compound_statement
        : lr declarations
                { nl(AT); }
          statements rr

declarations

declaration
        : int declarator_list sc

declarator_list
        : identifier
        | declarator_list co identifier

statements

statement
        : expression sc
        | sc
        | break sc
        | continue sc
        | return sc
        | return
                { out(" "); }
          expression sc
        | compound_statement
        | if_prefix statement
                { uncond(EX); }
        | if_prefix statement else statement
                { uncond(EX); }
        | loop_prefix statement
                { uncond(EX); }

if_prefix
        : if lp expression rp
                { cond(IN); }

loop_prefix
        : while lp expression rp
                { cond(IN); }
```

```
expression
        : binary
        | expression co binary

binary
        : identifier
        | constant
        | lp expression rp
        | identifier lp optional_argument_list rp
        | pp identifier                    %prec PP
        | mm identifier                    %prec PP
        | binary pl binary                 %prec '+'
        | binary mi binary                 %prec '+'
        | binary mu binary                 %prec '*'
        | binary di binary                 %prec '*'
        | binary rm binary                 %prec '*'
        | binary gt binary                 %prec '>'
        | binary lt binary                 %prec '>'
        | binary ge binary                 %prec '>'
        | binary le binary                 %prec '>'
        | binary eq binary                 %prec EQ
        | binary ne binary                 %prec EQ
        | binary an binary                 %prec '&'
        | binary xo binary                 %prec '^'
        | binary or binary                 %prec '|'
        | identifier as binary             %prec '='
        | identifier pe binary             %prec '='
        | identifier me binary             %prec '='
        | identifier te binary             %prec '='
        | identifier de binary             %prec '='
        | identifier re binary             %prec '='

optional_argument_list

argument_list
        : binary
        | argument_list co binary
```

```
/*
 *      Eingabesymbole ausgeben
 */

int     : INT           { out("int\t"); }
identifier : Identifier { out($1); }
lp      : '('           { out("("); }
rp      : ')'           { out(")"); }
co      : ','           { out(", "); }
sc      : ';'           { out(";"); nl(AT); }
break   : BREAK         { out("break"); }
continue: CONTINUE      { out("continue"); }
return  : RETURN        { out("return"); }
lr      : '{'           { cond(EX); out("{\t"); at(IN); }
rr      : '}'           { at(EX); out("}"); uncond(AT); }
if      : IF            { out("if "); }
else    : ELSE          { at(EX); out("else"); cond(IN); }
while   : WHILE         { out("while "); }
constant: Constant      { out($1); }
pp      : PP            { out(" ++ "); }
mm      : MM            { out(" -- "); }
pl      : '+'           { out(" + "); }
mi      : '-'           { out(" - "); }
mu      : '*'           { out(" * "); }
di      : '/'           { out(" / "); }
rm      : '%'           { out(" % "); }
gt      : '>'           { out(" > "); }
lt      : '<'           { out(" < "); }
ge      : GE            { out(" >= "); }
le      : LE            { out(" <= "); }
eq      : EQ            { out(" == "); }
ne      : NE            { out(" != "); }
an      : '&'           { out(" & "); }
xo      : '^'           { out(" ^ "); }
or      : '|'           { out(" | "); }
as      : '='           { out(" = "); }
pe      : PE            { out(" += "); }
me      : ME            { out(" -= "); }
te      : TE            { out(" *= "); }
de      : DE            { out(" /= "); }
re      : RE            { out(" %= "); }
```

%prec ist ein *yacc*-Operator, der einer Formulierung folgen kann. Dies ist eine ähnliche Vorschrift gegen Mehrdeutigkeit, wie wenn man Operatoren in **%left**-, **%right**- und **%nonassoc**-Anweisungen angibt. Auf **%prec** muß ein Eingabesym-

bol folgen, das den Vorrang definiert. Diese Konstruktion wird oft für unitäre Operatoren verwendet. Hier ist sie notwendig, da wir alle Operatoren durch Grammatikbegriffe ersetzt haben, damit wir die Ausgabeaktionen zusammenfassen können.

Wir illustrieren, wie die Formatieraktionen zusammenarbeiten, mit folgenden Beispielen:

```
if (1)
        1;
```

und

```
if (1)
{
        1;
}
```

In beiden Fällen wird **cond(IN)** bei der Reduzierung von **if_prefix** aufgerufen; dadurch wird der linke Rand nach innen verlegt. Im ersten Fall ist **statement** nur **expression** und der linke Rand bleibt verlagert. Im zweiten Fall ist **statement** ein **compound_statement** und deshalb wird **lr** für die linke geschweifte Klammer reduziert, bevor eine andere Reduzierung stattfindet. Bei der Reduzierung von **lr** wird **cond(EX)** aufgerufen. Diese Funktion stellt mit **condflag** fest, daß ein Aufruf von **cond(IN)** unmittelbar vorausging. Die linke Klammer kann daher noch nach links ausgestellt und direkt unterhalb von **if** angeordnet werden. Wie versprochen übertragen **static**-Variablen Information zwischen aufeinanderfolgenden Reduzierungen. Man kann diese Technik nur deshalb empfehlen, weil sie ziemlich abstruse Umformulierungen der Grammatik vermeiden hilft.

Die Formulierung von **compound_statement** zeigt eine zweite Art von Aktionen:

```
compound_statement
        : lr declarations
                { nl(AT); }
          statements rr
```

Unser Formatierstil schreibt vor, daß auf **declarations** eine leere Zeile folgt. Der Aufruf **nl(AT)** sorgt für diese leere Zeile. Man kann eine Aktion (in geschweiften Klammern) überall in einer Formulierung angeben. Steht sie nicht am Ende der Formulierung, generiert *yacc* einen anonymen Grammatikbegriff an Stelle der Aktion und definiert für diesen Begriff eine leere Formulierung, der dann die Aktion folgt. Die obige Formulierung wird in Wirklichkeit folgendermaßen expandiert:

```
compound_statement
        : lr declarations $$123 statements rr

$$123
        : /* leer */
                { nl(AT); }
```

Man kann Aktionen so nicht ganz willkürlich anordnen - die anonymen Grammatikbegriffe können für Konflikte sorgen.

Ein Problem haben wir ausgespart: dieser Formatierer bearbeitet nur eine Quelldatei, aus der die Kommentare schon entfernt wurden! Bei einer realistischen Lösung müßten die Kommentare in der lexikalischen Analyse gesammelt und, vermutlich an Eingabesymbole angefügt, der Formatierroutine übergeben werden. Dabei muß ziemlich viel Aufwand im Detail getrieben werden, außerdem ist eine sinnvolle Formatierung von Kommentaren selbst nicht einfach.

3.8 Aufgaben

1. Erweitern Sie den Tischrechner so, daß Variablen zur Verfügung stehen. Eine sehr einfache Lösung besteht darin, 26 Variablen mit Namen von **a** bis **z** samt Speicher für ihre Werte vorzudefinieren. Interessanter wird das Problem, wenn man beliebige Namen erlaubt. In diesem Fall sollte man Speicherplatz für die Variablennamen und für die Werte dynamisch verwalten. ⌨

2. Unser Formatierstil im Abschnitt 3.7 entspricht vielleicht nicht Ihren Vorstellungen. Ändern sie das Formatierprogramm so ab, daß es Ihrem Standard entspricht. ⌨

3. Schreiben Sie ein Formatierprogramm für eine Teilmenge von Pascal, vielleicht für die, die sie in Aufgabe 1 im Abschnitt 1.8 bearbeitet haben. ⌨

4. Der Formatierer im Abschnitt 3.7 gibt eine leere Zeile im Anschluß an jede Funktion aus. Dies bedeutet, daß bei einer **function_definition**, die als letzte **definition** in der Eingabe vorkommt, eine nutzlose leere Zeile am Schluß der Ausgabe steht. Ändern Sie den Formatierer, damit diese zusätzliche leere Zeile unterdrückt wird. Hinweis: es ist vielleicht leichter diese leere Zeile im Zaum zu halten, wenn sie *vor* einer Funktion ausgegeben wird. ⌨

5. Schreiben Sie einen Formatierer für EBNF. EBNF sollte in einem Standardformat ausgegeben werden, zum Beispiel so wie im Abschnitt 1.7. Vernünftigerweise beginnt man dazu mit der Lösung der Aufgabe 3 im Abschnitt 1.8. ⌨

6. Schreiben Sie ein Programm, das eine Grammatik aus EBNF in BNF übersetzt. Hinweis: aus Effizienzgründen innerhalb von *yacc* sollte man Wiederholungen linksrekursiv formulieren. Vergleichen Sie dazu die Diskussion zum *yacc*-Stack im Zusammenhang mit den links- und rechts-rekursiven Definitionen von

```
expression : expression '-' expression
```

im Abschnitt 3.4. ⌨

7. Ändern sie */usr/lib/yaccpar* so ab, daß die Ablaufverfolgung so ist, wie in diesem Kapitel gezeigt. Für wichtige Hinweise hierzu können Sie Abschnitt 8 und 9 im Anhang zu Rate ziehen. ⌨

8. Im letzten Absatz im letzten Abschnitt wurde erwähnt, daß die Übernahme von Kommentaren in die Ausgabe des Formatierers auf gewisse Schwierigkeiten stößt.

In dem Absatz wurde auch eine mögliche Technik vorgeschlagen. Ändern Sie das Beispiel aus diesem Abschnitt, oder ein Programm aus Aufgabe 2, 3 oder 4 so ab, daß Kommentare ausgegeben werden. Vorschlag: ist ein Kommentar das erste Objekt auf einer Zeile, dann sollte er auch am Anfang der Zeile (oder auch am aktuellen linken Rand) unverändert ausgegeben werden. Folgt der Kommentar anderen Objekten auf einer Zeile, sollten Sie ihn an einer global definierten Zeilenposition ausgeben, die vielleicht auch der Benutzer global setzen kann. ⇨

9. Mit ähnlichen Techniken wie in der vorigen Aufgabe können Sie ein Programm entwickeln, das auch Preprozessor-Zeilen ausgibt. Diese beginnen bekanntlich mit # in Spalte 1. ⇨

Kapitel 4: Fehlerbehandlung

Compiler müssen sich hauptsächlich mit fehlerhaften Eingabedateien beschäftigen. In diesem Kapitel sehen wir, wie wir unseren Parser gegen Eingabefehler widerstandsfähig machen können. Als erstes verwenden wir die *yacc*-Ablaufverfolgung um zu sehen, was im Parser passiert, wenn ein Fehler gefunden wird. Es zeigt sich, daß der bisher entwickelte Parser im Zuge einer **error**-Operation `vom Stack fallen´ würde. *yacc* verfügt jedoch über ein spezielles, reserviertes Eingabesymbol **error**, mit dem der Parser-Algorithmus beeinflußt werden kann. Im Abschnitt 4.2 demonstrieren wir dann, was passiert, wenn ein Parser ordentlich auf einen **error** vorbereitet ist.

Das allgemeine Problem besteht darin, mit Hilfe von **error**-Eingabesymbolen in einigen zusätzlichen Formulierungen robuste Grammatiken zu konstruieren. Erfreulicherweise kennen wir eine unkomplizierte Technik, mit der man einige Konstruktionen, die häufig in Programmiersprachendefinitionen vorkommen, so erweitern kann, daß sie mit *jedem* Fehler fertig werden. Wir führen unsere Technik im Abschnitt 4.3 vor, und im Abschnitt 4.5 zeigen wir dann, wie man einen robusten Parser für *kleinC* definiert, und wie man dessen Verhalten bei Eingabefehlern demonstrieren kann.

4.1 Problemstellung

Bisher kann unser Parser einen Satz, also eine *korrekte* Eingabedatei, erkennen und auch manipulieren. Ein Beispiel im Abschnitt 3.4 zeigte, was passiert, wenn man die fehlerhafte Eingabe

```
a - -
```

einem Parser liefert, der auf folgender Regel beruht:

```
%left   -
%%

expression
        : expression '-' expression
        | IDENTIFIER
```

yyparse() fällt vom Stack:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 3
```

```
[yydebug] reading '-'
[Fehler 1] Zeile 1 bei "-": erwartet wird: IDENTIFIER
[yydebug] recovery pops 3, uncovers 1
[yydebug] recovery pops 1, uncovers 0
[yydebug] recovery pops 0, stack is empty
yyparse() == 1
```

Das zweite – im Zustand 3 führt zu einer **error**-Operation in der Übergangsmatrix:

```
state 3
        expression :  expression -_expression

        IDENTIFIER  shift 2
        .  error

        expression  goto 4
```

Nach Ausgabe der Fehlermeldung entfernt **yyparse()** anscheinend alle Zustände vom Stack - offensichtlich auf der Suche nach ganz bestimmter Information. Da dabei der Stack total abgeräumt wird, geht der Aufruf von **yyparse()** mit einem Funktionswert von **1** zu Ende, und der Versuch einer Satzerkennung wird beim ersten Eingabefehler abgebrochen!

4.2 Elementare Techniken

Unsere Funktionen zur lexikalischen Analyse haben normalerweise folgende Zeile am Ende der Mustertabelle:

```
.                   return yytext[0];
```

Das Muster soll alle Operatoren finden, die aus jeweils einem einzelnen Zeichen bestehen. Das Muster liefert jedoch den Integer-Wert *jedes* einzelnen Zeichens als Funktionswert von **yylex()**, das heißt, als Eingabesymbol, wenn dieses Zeichen nicht vorher von einem anderen Muster akzeptiert wird.

Nicht vorgesehene Eingabezeichen werden daher von der lexikalischen Analyse an den Parser übergeben, als ob sie akzeptable Eingabesymbole wären, die als einzelne Zeichen repräsentiert sind. Dadurch werden alle Eingabefehler, auch die in der lexikalischen Analyse, gleichartig behandelt, nämlich im Parser. Alternativ kann natürlich die Funktion zur lexikalischen Analyse auch ihre eigenen Probleme berichten und dann derartige unerwünschte Zeichen einfach ignorieren. Bei diesem Verfahren kann man aber Kaskaden von Fehlermeldungen nur schwer vermeiden.

Zur Grammatik können wir Formulierungen hinzufügen, die zwar eigentlich nicht erlaubt, aber doch wahrscheinlich sind. Mit dieser Technik machen wir unseren Parser toleranter, als dies der Designer der Sprache wollte. Auf diese Weise können wir zwar die häufigsten Benutzerfehler abfangen, aber die Technik hat keine sehr hohen Aussichten insgesamt erfolgreich zu sein - eine fehlerhafte Eingabe kann man praktisch nie exakt voraussagen.

Wesentlich besser behandelt man einen Eingabefehler als Spezialfall eines Eingabesymbols: **error** ist ein vordefiniertes Eingabesymbol für *yacc*. **error** kann man in Formulierungen wie jedes andere Eingabesymbol verwenden. **error** wird jedoch (normalerweise) nicht als Funktionswert der lexikalischen Analyse produziert. Statt dessen nimmt der Parser selbst an, daß **error** das nächste Eingabesymbol ist, wenn das tatsächlich angebotene Eingabesymbol auf eine **error**-Operation in der Übergangsmatrix für den aktuellen Zustand führt. Wurde **error** als Eingabesymbol auf diese Weise intern generiert und wurde die obligate Fehlermeldung ausgegeben, versucht **yyparse() error** fast wie ein normales Eingabesymbol zu akzeptieren.

Betrachten wir einmal die folgende Modifikation der obigen Regel:

```
expression
        : expression '-' expression
        | IDENTIFIER
        | error
```

Ein aus dieser Grammatik erzeugter Parser akzeptiert fehlerhafte Eingaben stillschweigend. Um zu verstehen, wie das vor sich geht, müssen wir nochmals einige Beispiele verfolgen. *yacc* erzeugt folgende *y.output* Datei:

```
state 0
        $accept : _expression $end

        error  shift 3
        IDENTIFIER  shift 2
        .  error

        expression  goto 1

state 1
        $accept :  expression_$end
        expression :  expression_- expression

        $end  accept
        -  shift 4
        .  error

state 2
        expression :  IDENTIFIER_    (2)

        .  reduce 2
```

```
state 3
        expression :  error_    (3)

        .  reduce 3

state 4
        expression :  expression -_expression

        error  shift 3
        IDENTIFIER  shift 2
        .  error

        expression  goto 5

state 5
        expression :  expression_- expression
        expression :  expression - expression_    (1)

        .  reduce 1
```

Man beachte, daß eine **error**-Operation in einer Reihe von Zuständen auftaucht. Im Zustand 0 und 4 erscheint **error** auch als Eingabesymbol, für das die Operation **shift 3** in der Übergangsmatrix steht.

Sehen wir nun, wie dieser Parser die fehlerhafte Eingabe

```
a - - b
```

verarbeitet. Die Ablaufverfolgung zeigt, daß bis zum zweiten – Zeichen der Parser genau wie vorher funktioniert:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '-'
[yydebug] push state 4
[yydebug] reading '-'
[Fehler 1] Zeile 1 bei "-": erwartet wird: IDENTIFIER
```

Wir sind im Zustand 4, und das zweite – Zeichen hat zu einer **error**-Operation geführt. Wie gerade besprochen, nimmt **yyparse()** jetzt an, daß **error** das nächste Eingabesymbol ist. Die Übergangsmatrix schreibt vor, daß das **error**-Symbol akzeptiert werden soll und wir gehen in den Zustand 3 über:

```
[yydebug] accepting $error
[yydebug] push state 3
```

Wir können diesen Ablauf mit der Übergangsmatrix in *y.output* nachvollziehen: Im Zustand 3 ist eine Reduzierung möglich, aus **error** wird **expression** konstruiert und damit erreichen wir vom Zustand 4 den Zustand 5 auf dem Stack.

```
[yydebug] reduce by (3), uncover 4
[yydebug] push state 5
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
[yydebug] push state 4
```

Jetzt ist eine weitere Reduzierung möglich, wir erreichen Zustand 0 auf dem Stack, eine **goto**-Operation für die gerade reduzierte **expression** führt uns in den Zustand 1 und dort können wir schließlich das – Zeichen aus der Eingabe akzeptieren! Jetzt verläuft alles wieder ganz normal und die Erkennung wird erfolgreich beendet:

```
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 4
[yydebug] push state 5
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
[yydebug] reading [end of file]
yyparse() = 0
```

Die Fehlerbehandlung bestand tatsächlich darin, daß das **error**-Symbol zwischen den zwei – Zeichen in der Eingabe *eingefügt* wird. Die daraus resultierende Eingabe war im Sinne der erweiterten Grammatik akzeptabel. Das ist aber noch nicht alles - manchmal muß bei der Fehlerbehandlung ein Eingabesymbol auch *ignoriert* werden, damit die Erkennung erfolgreich verlaufen kann. Dazu wird **error** von *yacc* und **yyparse()** ganz anders als normale Eingabesymbole behandelt. Schauen wir an, was passiert, wenn

```
a + - b
```

bearbeitet werden soll. Hier ist + ein illegales Zeichen, das von **yylex()** wie ein akzeptables Eingabesymbol geliefert wird. Dieses Symbol muß ganz bestimmt ignoriert werden:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 0
[yydebug] push state 1
[yydebug] reading '+'
[Fehler 1] Zeile 1 bei "+": erwartet wird: '-'
```

Diesmal sind wir im Zustand 1 in Schwierigkeiten. Wieder wird intern **error** als Eingabesymbol generiert, aber anders als im Zustand 4 enthält im Zustand 1 die Über-

gangsmatrix eine **error**-Operation für das **error**-Symbol! (In *y.output* bedeutet . `alle anderen Symbole'.)

Jetzt beginnt **yyparse()** seinen Stack abzuräumen. Der Parser sucht rückwärts *auf dem Stack* nach einem Zustand, in dem das **error**-Symbol akzeptiert werden kann. Findet sich kein solcher Zustand auf dem Stack, das heißt, haben wir vorher noch keine Konfiguration erreicht, bei der **error** das nächste Symbol war, räumt **yyparse()** den ganzen Stack ab und terminiert dann mit einem Funktionswert **1**.

In unserem Beispiel haben wir `Glück'. Wir finden Zustand 0 auf dem Stack und in diesem Zustand kann **error** akzeptiert werden. Wieder erfolgt eine **shift**-Operation in den Zustand 3, **expression** wird aus **error** konstruiert, wir kehren auf dem Stack zum Zustand 0 zurück und erreichen den Zustand 1 mit einer **goto**-Operation für **expression**:

```
[yydebug] recovery pops 1, uncovers 0
[yydebug] accepting $error
[yydebug] push state 3
[yydebug] reduce by (3), uncover 0
[yydebug] push state 1
```

Es sieht ganz so aus, als ob wir genauso weit sind wie vorher: wir sind wieder im Zustand 1 und **+** ist noch immer das nächste Eingabesymbol! Jetzt erinnert sich aber **yyparse()**, daß seit der letzten **error**-Operation noch keine **shift**-Operation stattgefunden hat. Um eine Schleife zu vermeiden, ignoriert **yyparse()** jetzt dieses nächste Eingabesymbol und kann bei unserem Beispiel dann die Erkennung erfolgreich abschließen:

```
[yydebug] recovery discards '+'
[yydebug] reading '-'
[yydebug] push state 4
[yydebug] reading IDENTIFIER
[yydebug] push state 2
[yydebug] reduce by (2), uncover 4
[yydebug] push state 5
[yydebug] reduce by (1), uncover 0
[yydebug] push state 1
[yydebug] reading [end of file]
yyparse() = 0
```

Fügen wir einmal beide Fehler zusammen. Ohne Ablaufverfolgung liefert die Eingabe

```
a - - b + - c
```

nur eine einzige Fehlermeldung:

```
[Fehler 1] Zeile 1 bei "-": erwartet wird: IDENTIFIER
yyparse() = 0
```

Die Ablaufverfolgung zeigt, daß schon beide Fehler erkannt werden:

```
[yydebug] push state 0
[yydebug] reading IDENTIFIER
        ...
[yydebug] reading '-'
[Fehler 1] Zeile 1 bei "-": erwartet wird: IDENTIFIER
[yydebug] accepting $error
        ...
[yydebug] reading '+'
[yydebug] recovery pops 1, uncovers 0
[yydebug] accepting $error
        ...
[yydebug] recovery discards '+'
        ...
yyparse() = 0
```

Um eine Kaskade von Fehlermeldungen zu vermeiden, muß der Parser im Anschluß an ein **error**-Symbol wenigstens drei **shift**-Operationen für Eingabesymbole durchführen, bevor ein weiterer Fehler auch berichtet wird. Dadurch kann eine Häufung von Eingabefehlern möglicherweise nur zu einer einzigen Fehlermeldung führen. Hier erklärt dies das Fehlen der zweiten Fehlermeldung.

Mit einer **yyerrok**-Aktion kann man dem Parser einreden, daß er genügend Eingabesymbole in einer solchen Situation akzeptiert hat, und daß er folglich auch nahe benachbarte Fehler einzeln berichtet.

Es gibt dabei allerdings eine Falle: wird **yyerrok** als Aktion einer Formulierung zugeordnet, die nur aus **error** besteht, glaubt **yyparse()** sofort, daß genügend Eingabesymbole akzeptiert wurden; in diesem Fall kann **yyparse()** ein fehlerhaftes Eingabesymbol aber nicht ignorieren!

Ein sinnvolleres Beispiel für eine **yyerrok**-Aktion ist folgende Erweiterung unserer Grammatik:

```
expression
        : expression '-' expression
        | IDENTIFIER
                { yyerrok; }
        | error
```

Haben wir **IDENTIFIER** im Anschluß an **error** erkannt, ist es plausibel, daß unser Parser wieder korrekt operiert; dann ist es auch sinnvoll, daß wir wieder Fehlermeldungen sehen wollen. Mit dieser Korrektur produzieren wir jetzt zwei Fehlermeldungen in unserem Beispiel:

```
[Fehler 1] Zeile 1 bei "-": erwartet wird: IDENTIFIER
[Fehler 2] Zeile 1 bei "+": erwartet wird: '-'
yyparse() = 0
```

error-Eingabesymbol und **yyerrok**-Aktion sind die wesentlichen Bestandteile von *yacc*, mit denen man einen Parser gegen Eingabefehler robust macht. Der Trick besteht darin, sie auch gekonnt einzusetzen.

4.3 Plazierung der "error"-Symbole

Bei der Plazierung von **error**-Symbolen in der Grammatik verfolgen wir folgende, einander widersprechende Ziele:

- so nahe wie möglich zum Startsymbol der Grammatik.

  Dadurch gibt es immer einen Punkt, an dem die Erkennung wieder aufgesetzt werden kann, denn dann sollte sich immer weit unten auf dem Stack ein Zustand befinden, in dem **error** akzeptiert werden kann. Damit kann der Parser den Stack nie völlig abräumen, das heißt, die Erkennung kann nie vor Erreichen des Dateiendes in der lexikalischen Analyse abgebrochen werden.

- so nahe wie möglich zu jedem Eingabesymbol.

  Damit wird bei jedem Fehler nur wenig von der Eingabe übersprungen. Dazu kann man auch noch mit **yyerrok**-Aktionen nachhelfen.

- ohne daß Konflikte in der Grammatik entstehen.

  Das kann recht schwierig sein. Tatsächlich kann man *shift/reduce*-Konflikte durchaus akzeptieren, so lange sie dazu führen, daß längere Folgen von Eingabesymbolen zusammengefaßt werden. Man kann zum Beispiel versuchen, **expression** über einen Fehler hinweg fortzusetzen, anstatt den gleichen Fehler erst innerhalb von **statement** zu akzeptieren und damit den Rest des arithmetischen Ausdrucks zu ignorieren.

In Sinne dieser Ziele empfehlen wir folgende typische Positionen für **error**-Symbole in einer Grammatik:

- in jeder rekursiven Konstruktion, das heißt, in jeder Wiederholung.

- vorzugsweise nicht am Ende einer Formulierung.

  Dadurch sollte sich eine robuste Fehlerbehandlung ergeben, das heißt, die Fortsetzung der Erkennung sollte jeweils sinnvoll sein. Fügt man **error** und eine **yyerrok**-Aktion am Ende einer Formulierung hinzu, ergeben sich leicht Kaskaden von Fehlermeldungen, oder sogar Schleifen, wenn der Parser dann Eingabesymbole nicht mehr ignorieren kann.

- Listen mit wenigstens einem Element benötigen zwei **error**-Formulierungen: eine, die ein Problem am Anfang der Liste behandelt, und eine, die für ein Problem am momentanen Ende der Liste zuständig ist.

- Listen, die auch leer sein können, benötigen ein **error**-Symbol in der leeren Formulierung.

Sollte dies nicht möglich sein, muß man **error** überall dort einfügen, wo die möglicherweise leere Liste verwendet wird.

Die folgende Tabelle ist unsere Empfehlung für die Plazierung von **error**-Symbolen in den typischen Konstruktionen zur Wiederholung: [1]

Konstruktion	EBNF	*yacc*-Eingabe

optionale Folge	`x: { y }`	`x: /* leer */`
		`| x y { yyerrok; }`
		`| x error`

Folge	`x: y { y }`	`x: y`
		`| x y { yyerrok; }`
		`| error`
		`| x error`

Liste	`x: y { T y }`	`x: y`
		`| x T y { yyerrok; }`
		`| error`
		`| x error`
		`| x error y { yyerrok;}`
		`| x T error`

Wir demonstrieren die drei Fälle der Reihe nach. In jedem Fall verwenden wir die gleiche lexikalische Analyse wie bei dem Tischrechner im Abschnitt 3.5. Bei der optionalen Folge kann mit folgender Eingabe für *yacc* die Fehlerbehandlung untersucht werden:

```
%{
#include <stdio.h>
#define put(x)  printf("%d ", x)
#define err(x)  fputs("err x ", stdout)
%}
%token  Constant
%%
```

[1] Ein Fehler in *yacc* (wie mit Bell Version 7, Berkeley 4.2bsd, und verschiedenen Abarten ausgeliefert) wirkt sich bei dieser Plazierung allerdings sehr negativ aus: findet in einem Zustand eine Reduzierung als Operation für 'alle anderen Symbole' statt, und kann das nächste Eingabesymbol nicht akzeptiert werden, kann aber **error** akzeptiert werden (zum Beispiel wenn **error** am Ende einer Formulierung steht), diktieren die Tabellen in *yacc*, daß die Reduzierung auch dann stattfindet, wenn anschließend das nächste Eingabesymbol nicht akzeptiert werden kann. In diesem Fall findet die Fehlerbehandlung 'zu spät' statt, der Parser kann tatsächlich in eine Schleife geraten, versehentlich Regeln mehrfach reduzieren, usw. In der 4.1bsd Implementierung von UNIX wurde *yacc* nach [Gra79] korrigiert, um diesen Fehler zu unterdrücken. Die Korrektur besteht darin, in solchen Fällen alle möglichen Eingabesymbole aufzuzählen, damit der Fehler sofort erkannt werden kann. Die Parser-Tabellen werden dadurch etwas größer. Leider enthält die Korrektur in 4.1bsd aber einen Tippfehler. Eine definitive Korrektur kann von den Autoren bezogen werden. (S. Johnson, persönliche Mitteilung, 1982).

```
line
        : /* leer */
        | line optional_sequence '\n'
                { putchar('\n');
                  yyerrok;
                }

optional_sequence
        : /* leer */
        | optional_sequence Constant
                { put($2);
                  yyerrok;
                }
        | optional_sequence error
                { err(1); }

%%

main()
{       extern FILE * yyerfp;

        yyerfp = stderr;        /* separates Protokoll */
        printf("yyparse() = %d\n", yyparse());
}
```

Wir setzen **stderr** als Filepointer für Fehlermeldungen (Abschnitt 3.3), damit wir die Fehlermeldungen getrennt von der Ausgabe der Aktionen erhalten. Die Aktionen sind so entworfen, daß sie zeigen, welche Reduzierungen der Parser vornimmt. Die Eingabe

```
10 20

10 +
10 + 20
```

liefert die Ausgabe

```
10 20

10 err 1
10 err 1 20
yyparse() = 0
```

und die Fehlermeldungen

```
[Fehler 1] Zeile 3 bei "+": erwartet wird: '\n' Constant
[Fehler 2] Zeile 4 bei "+": erwartet wird: '\n' Constant
```

Alle Eingabesymbole werden korrekt reduziert, obgleich Eingabefehler vorliegen.

Soll die Folge wenigstens ein Element enthalten, müssen wir den Parser geringfügig ändern:

```
sequence
        : Constant
                { put($1); }
        | sequence Constant
                { put($2);
                  yyerrok;
                }
        | error
                { err(1); }
        | sequence error
                { err(2); }
```

Die gleiche Eingabe ergibt

```
10 20
err 1
10 err 2
10 err 2 20
yyparse() = 0
```

und eine zusätzliche Fehlermeldung für die leere Zeile:

```
[Fehler 1] Zeile 2: erwartet wird: Constant
[Fehler 2] Zeile 3 bei "+": erwartet wird: '\n' Constant
[Fehler 3] Zeile 4 bei "+": erwartet wird: '\n' Constant
```

Wieder wurden alle Eingabesymbole korrekt reduziert.

Die Liste, eine Folge mit wenigstens einem Element und mit Trennern zwischen je zwei Elementen, hat mehr Fehlermöglichkeiten:

```
list
        : Constant
                { put($1); }
        | list ',' Constant
                { put($3);
                  yyerrok;
                }
        | error
                { err(1); }
        | list error
                { err(2); }
```

```
        | list error Constant
                { err(3);
                  put($3);
                  yyerrok;
                }
        | list ',' error
                { err(4); }
```

Die Testdaten reflektieren die zusätzliche Struktur der Liste:

```
10 , 20

10 +
10 20
10 ,
10 + 20
```

Die Ausgabe

```
10 20
err 1
10 err 2
10 err 3 20
10 err 4
10 err 2
yyparse() = 0
```

und die Fehlermeldungen

```
[Fehler 1] Zeile 2: erwartet wird: Constant
[Fehler 2] Zeile 3 bei "+": erwartet wird: '\n' ','
[Fehler 3] Zeile 4 bei "20": erwartet wird: '\n' ','
[Fehler 4] Zeile 5: erwartet wird: Constant
[Fehler 5] Zeile 6 bei "+": erwartet wird: '\n' ','
```

zeigen, daß wir in allen Fällen die Erkennung fortsetzen können. Leider wird im Fall

```
10 + 20
```

zur Fehlerbehandlung die Regel

```
list : list error
```

verwendet, und das zweite Element der Liste wird ignoriert! Eliminieren wir aber diese Formulierung, so geht die Erkennung bei einem Fehler am Schluß der Eingabe nicht korrekt zu Ende.

Unsere Empfehlungen zur Plazierung von **error**-Symbolen garantieren nicht, daß ein sinnvolles Eingabesymbol in manchen Fehlersituationen nicht doch ignoriert wird. Wir haben uns aber in der Praxis davon überzeugt, daß diese Empfehlungen systematisch zu sehr robusten Parsern für häufige Konstruktionen in Programmiersprachen führen.

4.4 Plazierung der "yyerrok"-Aktionen

yyerrok sollte man im Anschluß an Eingabesymbole überall dort angeben, wo eine Formulierung mit **error** aufhören kann, und wo ihr dann ein einigermaßen wichtiges Eingabesymbol folgt. Bei den oben beschriebenen Konstruktionen zur Wiederholung sind die relevanten Aktionen schon angegeben worden.

Wird das Eingabesymbol reduziert, so wird in diesem Fall ein Fehler anschließend sofort wieder berichtet - unabhängig von der Drei-Symbol-Regel.

Einige Symbole werden daher als sehr wichtig angesehen, in *kleinC* zum Beispiel:

```
sc      ;
rp      )
rr      }
```

Komma wird nicht in diese Liste aufgenommen, denn im Rahmen von Wiederholungen muß ihm **error** nicht unbedingt folgen; dies wird im Rahmen der Wiederholungen behandelt.

Für diese wichtigen Symbole konstruieren wir einfach Regeln mit Aktionen für Eingabesymbole, wie wir das im Formatierer schon für alle Eingabesymbole gemacht haben.

4.5 Beispiel

Wir fügen **error**-Symbole und **yyerrok**-Aktionen zur *kleinC*-Grammatik aus Abschnitt 1.6 genau nach Maßgabe der Tabelle aus Abschnitt 4.3 hinzu. Darüberhinaus ersetzen wir ';', ')' und '}' durch Grammatikbegriffe wie in Abschnitt 4.4 besprochen, damit diese Symbole immer **yyerrok**-Aktionen auslösen. Wird das Resultat mit *yacc* bearbeitet, finden wir zwei *shift/reduce*- und zwei *reduce/reduce*-Konflikte. Die Datei *y.output* zeigt folgende Probleme:

```
40: shift/reduce conflict (shift 46, red'n 39) on error
state 40
        compound_statement :  { declarations_statements rr
        declarations :  declarations_declaration
        declarations :  declarations_error
        statements : _    (39)

        error  shift 46
        Identifier  reduce 39
                ...
        .  error

        declaration  goto 45
        statements  goto 44
```

```
53: reduce/reduce conflict (red'ns 41 and 56 ) on error
53: reduce/reduce conflict (red'ns 41 and 56 ) on ;
state 53
        statements :  statements error_    (41)
        expression :  error_    (56)

        ,  reduce 56
        .  reduce 41

83: shift/reduce conflict (shift 115, red'n 49) on ELSE
state 83
        statement :  if_prefix statement_    (49)
        statement :  if_prefix statement_ELSE statement

        ELSE  shift 115
        .  reduce 49
```

Der *shift/reduce*-Konflikt im Zustand 38 rührt davon her, daß **declarations** und **statements** beide optionale Folgen sind, das heißt, leer sein, nur aus **error** bestehen oder **error** am Anfang oder Ende enthalten können. Wenn wir den Konflikt erlauben, machen wir **declarations** so lang wie möglich und das scheint wünschenswert.

Der zweite *shift/reduce*-Konflikt hat mit **else** zu tun und war in der ursprünglichen Grammatik schon vorhanden.

Die zwei *reduce/reduce*-Konflikte haben damit zu tun, daß das Ende einer **expression** nicht mehr korrekt erkannt wird. Nachdem wir

```
expression
        : error
```

durch

```
expression
        : error ',' binary
        | error binary
```

ersetzen, verschwinden diese *reduce/reduce*-Konflikte zwar plangemäß, aber wir handeln uns fünf neue *shift/reduce*-Konflikte ein! *y.output* zeigt warum:

```
53: shift/reduce conflict (shift 66, red'n 41) on Identifier
53: shift/reduce conflict (shift 67, red'n 41) on Constant
53: shift/reduce conflict (shift 68, red'n 41) on (
53: shift/reduce conflict (shift 69, red'n 41) on PP
53: shift/reduce conflict (shift 70, red'n 41) on MM
state 53
        statements :  statements error_    (41)
        expression :  error_, binary
        expression :  error_binary

        Identifier  shift 66
        Constant  shift 67
        (  shift 68
        PP  shift 69
        MM  shift 70
        ,  shift 75
        .  reduce 41

        binary  goto 76
```

Wir müssen darauf bestehen, daß vor **binary** ein Eingabesymbol steht. Wenn wir also bei allen relevanten Formulierungen für **expression** und aus Analogiegründen auch für **argument_list** darauf bestehen, daß ein Komma vorhanden ist, verschwinden diese neuen *shift/reduce*-Konflikte wieder.

Zum Schluß fügen wir noch ein paar **error**-Symbole hinzu, um an einigen Stellen die Reaktion des Parsers zu verbessern:

```
if_prefix
        : IF error

loop_prefix
        : WHILE error

binary
        : '(' error rp
```

Auch die **return**-Anweisung könnte entsprechend verfeinert werden.

Um nun zu zeigen, daß die Fehlerbehandlung erfolgreich verläuft, fügen wir zu jeder Formulierung mit **error** eine entsprechende Aktion hinzu. Die geänderten Teile der Grammatik sehen nun wie folgt aus:

```
/*
 *      kleinC
 *      Syntaxanalyse und Fehlerbehandlung
 *      (s/r Konflikte: bei ELSE, bei error)
 */

%{
#define ERROR(x) yywhere(), puts(x)
%}

/*
 *      Eingabesymbole
 */

/*
 *      Vorrangtabelle
 */

%%

program

definitions
        : definition
        | definitions definition
                { yyerrok; }
        | error
                { ERROR("definitions: error"); }
        | definitions error
                { ERROR("definitions: definitions error"); }

definition

function_definition
        : Identifier '(' optional_parameter_list rp
          parameter_declarations compound_statement

optional_parameter_list

parameter_list
        : Identifier
        | parameter_list ',' Identifier
                { yyerrok; }
        | error
                { ERROR("parm_list: error"); }
```

```
        | parameter_list error
                { ERROR("parm_list: parm_list error"); }
        | parameter_list error Identifier
                { ERROR("parm_list: parm_list error Id");
                  yyerrok;
                }
        | parameter_list ',' error
                { ERROR("parm_list: parm_list ',' error"); }

parameter_declarations
        : /* null */
        | parameter_declarations parameter_declaration
                { yyerrok; }
        | parameter_declarations error
                { ERROR("parm_decls: parm_decls error"); }

parameter_declaration
        : INT parameter_declarator_list sc

parameter_declarator_list
        : Identifier
        | parameter_declarator_list ',' Identifier
                { yyerrok; }
        | error
                { ERROR("parm_decl_l: error"); }
        | parameter_declarator_list error
                { ERROR("parm_decl_l: parm_decl_l error"); }
        | parameter_declarator_list error Identifier
                { ERROR("parm_decl_l: parm_decl_l error Id");
                  yyerrok;
                }
        | parameter_declarator_list ',' error
                { ERROR("parm_decl_l: parm_decl_l ',' error"); }

compound_statement
        : '{' declarations statements rr

declarations
        : /* null */
        | declarations declaration
                { yyerrok; }
        | declarations error
                { ERROR("declarations: declarations error"); }
```

```
declaration
        : INT declarator_list sc

declarator_list
        : Identifier
        | declarator_list ',' Identifier
                { yyerrok; }
        | error
                { ERROR("decl_list: error"); }
        | declarator_list error
                { ERROR("decl_list: decl_list error"); }
        | declarator_list error Identifier
                { ERROR("decl_list: decl_list error Id");
                  yyerrok;
                }
        | declarator_list ',' error
                { ERROR("decl_list: decl_list ',' error"); }

statements
        : /* null */
        | statements statement
                { yyerrok; }
        | statements error
                { ERROR("statements: statements error"); }

statement
        : expression sc
        | sc
        | BREAK sc
        | CONTINUE sc
        | RETURN sc
        | RETURN expression sc
        | compound_statement
        | if_prefix statement
        | if_prefix statement ELSE statement
        | loop_prefix statement

if_prefix
        : IF '(' expression rp
        | IF error
                { ERROR("if_prefix: IF error"); }

loop_prefix
        : WHILE '(' expression rp
        | WHILE error
                { ERROR("loop_prefix: WHILE error"); }
```

```
expression
        : binary
        | expression ',' binary
                { yyerrok; }
        | error ',' binary
                { ERROR("expression: error ',' binary");
                  yyerrok;
                }
        | expression error
                { ERROR("expression: expression error"); }
        | expression ',' error
                { ERROR("expression: expression ',' error"); }

binary
        : Identifier
        | Constant
        | '(' expression rp
        | '(' error rp
                { ERROR("binary: '(' error ')'"); }
        | Identifier '(' optional_argument_list rp
        | PP Identifier
        | MM Identifier
        | binary '+' binary
        | binary '-' binary
        | binary '*' binary
        | binary '/' binary
        | binary '%' binary
        | binary '>' binary
        | binary '<' binary
        | binary GE binary
        | binary LE binary
        | binary EQ binary
        | binary NE binary
        | binary '&' binary
        | binary '^' binary
        | binary '|' binary
        | Identifier '=' binary
        | Identifier PE binary
        | Identifier ME binary
        | Identifier TE binary
        | Identifier DE binary
        | Identifier RE binary

optional_argument_list
```

```
argument_list
        : binary
        | argument_list ',' binary
                { yyerrok; }
        | error
                { ERROR("arg_list: error"); }
        | argument_list error
                { ERROR("arg_list: arg_list error"); }
        | argument_list ',' error
                { ERROR("arg_list: arg_list ',' error"); }

/*
 *      bestimmte Eingabesymbole nie ignorieren
 */

rp      : ')'   { yyerrok; }
sc      : ';'   { yyerrok; }
rr      : '}'   { yyerrok; }
```

Folgende Eingabedatei probiert einige Aspekte der Fehlerbehandlung in dieser Grammatik aus:

```
f1() { }
char x;         /* 2: fehlerhafte Definition */
char y;         /* diese wird ignoriert -- kein yyerrok */
f2() { }        /* dies wird wieder parsiert */

f3(a,
        int,    /* 7: fehlerhafter Parameter */
        c) { }

f4(int          /* 10: fehlerhafter Parameter */
        ) { }

f5(a, b)
        int a;
        while ; /* 15: fehlerhafte Deklaration */
        int b;
        { }

int a,
        while,  /* 20: fehlerhafter Deklarator */
        b;
```

```
f6() {
        break   /* 24: fehlerhaftes Statement */
        break;
        return;
        }

f7() {
        a,
        int,    /* 31: fehlerhafte Expression */
        b;
        }

f8() {
        f7(a,
        int,    /* 37: fehlerhaftes Argument */
        b);
        }
```

Hier ist die zugehörige Ausgabe:

```
[Fehler 1] Zeile 2 bei "x": erwartet wird: '('
Zeile 2 bei "x": definitions: definitions error
Zeile 2 bei ";": definitions: definitions error
Zeile 3 bei "y": definitions: definitions error
Zeile 3 bei ";": definitions: definitions error
[Fehler 2] Zeile 8 bei "int": erwartet wird: Identifier
Zeile 8 bei "int": parm_list: parm_list ',' error
[Fehler 3] Zeile 11 bei "int": erwartet wird: Identifier
Zeile 11 bei "int": parm_list: error
[Fehler 4] Zeile 16 bei "while": erwartet wird: '{' INT
Zeile 16 bei "while": parm_decls: parm_decls error
[Fehler 5] Zeile 21 bei "while": erwartet wird: Identifier
Zeile 21 bei "while": decl_list: decl_list ',' error
[Fehler 6] Zeile 26 bei "break": erwartet wird: ';'
Zeile 26 bei "break": statements: statements error
[Fehler 7] Zeile 32 bei "int": erwartet wird: '('
                                        Identifier Constant PP MM
Zeile 32 bei "int": expression: expression ',' error
[Fehler 8] Zeile 38 bei "int": erwartet wird: '('
                                        Identifier Constant PP MM
Zeile 38 bei "int": arg_list: arg_list ',' error
```

Die Ausgabe zeigt, daß einige nahe beieinanderliegende Fehler nicht einzeln berichtet werden.

4.6 Aufgaben

1. Erzeugen Sie eine Eingabedatei, die bei der Version von *kleinC* aus Abschnitt 4.5 jede Formulierung mit **error** ausprobiert. ✏

2. Im Beispiel werden mehrere Fehler bei Definitionen als ein Fehler berichtet. Wie kann man dies besser machen? Probieren Sie Ihre Lösung mit einer Eingabedatei aus, die eine Folge von geeigneten Fehlern enthält. ✏

3. Installieren Sie Fehlerbehandlung im Tischrechner aus Abschnitt 3.5, oder in dem, den Sie für Aufgabe 1 im Abschnitt 3.8 entwickelt haben. Hinweis: für **line** wurde die prinzipielle Technik im Abschnitt 4.3 vorgeführt. Für **expression** eine Fehlerbehandlung durchzuführen ist ein interessanteres Problem. ✏

4. Fügen Sie die für **expression** in Aufgabe 3 entwickelte Lösung zur Fehlerbehandlung von *kleinC* hinzu. ✏

5. Ändern Sie den Tischrechner aus Ausgabe 3 so ab, daß er nach jedem Fehler den Benutzer dazu auffordert, eine korrigierte Zeile einzugeben. Hinweis: die **yyerrok**-Aktionen müssen sorgfältig plaziert werden, denn der Parser darf natürlich nicht Symbole von der korrigierten Zeile im Zuge der Fehlerbehandlung für die ursprüngliche Zeile einfach ignorieren. ✏

6. Fügen Sie Fehlerbehandlung zur Pascal-Grammatik hinzu, die Sie in Aufgabe 1 im Abschnitt 1.8 konstruiert haben. Probieren Sie das Resultat mit einer Eingabedatei aus, die ausreichend fehlerhaften Pascal-Text enthält. ✏

Kapitel 5: Semantikprüfung

Vom allgemeinen Problem der Konstruktion stabiler Parser ausgehend wenden wir uns jetzt dem spezielleren Problem zu, Programmtext zu analysieren um eine übersetzte, ausführbare Version zu produzieren. Wir verwenden deshalb jetzt den Begriff *Programm* an Stelle von *Satz*, das heißt, ein Programm ist als Folge von Eingabesymbolen definiert, für die ein eindeutiger Syntaxbaum in bezug auf eine Grammatik konstruiert werden kann. Dieses Kapitel beschreibt, wie wir weitere Einschränkungen für ein Programm festlegen können. Damit wird die *Analyse* im Compiler vervollständigt. In den folgenden Kapiteln betrachten wir dann die *Synthese* einer 'ausführbaren Version des Algorithmus', den ein Programm beschreibt.

5.1 Problemstellung

Ein Programm kann syntaktisch korrekt sein und trotzdem semantische Fehler enthalten. Typische Beispiele sind etwa folgende:

> In Pascal sind Marken Ziffernketten, die in einer **label**-Deklaration vereinbart werden, bevor sie benutzt werden können. Bei der lexikalischen Analyse wird eine derartige Ziffernkette vielleicht als **IntegerConstant** erkannt; wird sie jedoch in einer **goto**-Anweisung verwendet, muß der Compiler überprüfen, daß sie auch wirklich als Marke vereinbart wurde.
>
> In Pascal müssen Marken deklariert werden, bevor sie benützt werden können. In C ist eine Marke schlicht ein **Identifier**. Taucht ein neuer Name im Anschluß an **goto** auf, wird er implizit als Marke vereinbart. In fast allen Sprachen können Marken auch erst *definiert* werden *nachdem* sie bereits benutzt wurden. Für alle Marken muß ein Compiler folglich verifizieren, daß sie auch wirklich definiert wurden.

Marken sind nur ein - unangenehmeres - Beispiel für *Geltungsbereich*-Probleme: ein vom Benutzer eingeführtes Objekt ist nur in einem bestimmten Teil des Programmtexts bekannt, eben im sogenannten *Geltungsbereich* eines Namens. In Basic erstreckt sich der Geltungsbereich eines Variablennamens (mit Ausnahme der Parameternamen in Benutzerfunktionen) auf den ganzen Programmtext. In Fortran sind Variablennamen nur in einem Programmteil bekannt, das heißt, in einer **function** oder **subroutine**; die Namen dieser Programmteile und die Namen von **common**-Bereichen sind in allen Modulen bekannt, die zusammen in ein *image* eingebunden werden, das heißt, in eine Datei, die ausgeführt werden kann. In Pascal und anderen Sprachen aus der Algol-Familie sind vom Benutzer eingeführte Namen nur innerhalb von einem *Block* bekannt, das heißt, innerhalb von einem syntaktisch begrenzten Bereich des Programmtexts, der eine Vereinbarung für den Namen enthält. Blöcke können verschachtelt werden, und die Definition eines Namens in einem äußeren Block kann in einem inneren Block verborgen werden durch eine neue Vereinbarung für diesen Namen im inneren Block. C kombiniert die von Algol bekannte Blockstruktur mit dem Modulkonzept von Fortran: *compound-statements* sind Blök-

ke, die verschachtelt werden können, Vereinbarungen enthalten können, und die den Geltungsbereich dieser Vereinbarungen begrenzen; Funktionsnamen sind global bekannt und müssen nicht unbedingt vor Gebrauch vereinbart werden. Ein Compiler muß sowohl die korrekte Verwendung der vom Benutzer eingeführten Namen innerhalb der jeweiligen Geltungsbereiche überwachen, als auch Code generieren, der entsprechenden Zugriff auf die verschiedenen Objekte ermöglicht.

Im gleichen Kontext kann man Namen im allgemeinen nicht doppelt vereinbaren, beispielsweise können zwei Parameter nicht den gleichen Namen haben, zwei lokale Variablen im gleichen Block müssen verschiedene Namen besitzen, wie auch zwei Komponenten in der gleichen Struktur, also in einem **struct**-, **union**- oder **record**-Bereich. In C kann zwar eine Struktur den gleichen Namen besitzen wie eine andere Variable im gleichen Geltungsbereich, aber manche C-Compiler verlangen sogar, daß Komponentennamen selbst zwischen verschiedenen Strukturen verschieden sein müssen.

Vereinbarungen in einem Programm dienen dazu, dem Compiler die beabsichtigte Verwendung des Namens mitzuteilen. Besteht Einigkeit über die Verwendung, muß Mißbrauch verhindert werden. Zum Beispiel:

In den meisten Sprachen - mit der besonderen Ausnahme von PL/I und von bewußt toleranten Werkzeugen wie *awk* - dürfen Zeichenketten und arithmetische Werte nicht direkt zusammengefaßt werden, etwa für Addition. In Pascal sind *mixed-mode*-Ausdrücke, das heißt, Kombinationen von **real**- und **integer**-Werten in arithmetischen Ausdrücken, bei den meisten Operatoren erlaubt, aber es gibt da gewisse Einschränkungen: **div** beschreibt eine reine **integer**-Division, / liefert ein **real**-Resultat sogar für zwei **integer**-Operanden, := erlaubt Zuweisung von **integer** an **real** aber nicht umgekehrt, usw.

Die exakte Bedeutung eines Operators hängt vom Typ seiner Operanden ab. In einigen Basic Dialekten, zum Beispiel, bezeichnet + Addition bei numerischen Werten und Verkettung bei Zeichenketten. In Pascal bezeichnet + Addition bei numerischen Werten und Vereinigung bei **set**-Werten - wenigstens bei der üblichen Wahl von Eingabesymbolen für diese Sprache. In C beschreibt + komplizierte Adreßoperationen, wenn ein Zeiger und ein Integer-Wert als Operanden verwendet werden. Numerisch, und als Maschinenbefehl, ist + als Operation zwischen Integer-Werten sehr verschieden von der Addition von Gleitkommawerten: bei Integer-Werten hängt das Resultat von der Reihenfolge der Werte und von implizit eingefügten Klammern nicht ab, ganz im Gegensatz zur Gleitkommaoperation.

Die Auswahl von Komponenten bei **struct**, **union** oder **record** bedingt im allgemeinen, daß der Komponentenname auch zur Struktur der Variablen gehört, auf die die Operation angewendet wird, das heißt, für Operatoren wie ., ∧ und -> gibt es ziemlich strenge Bedingungen in bezug auf die Typen ihrer Operanden. C ist hier - absichtlich - sehr nachsichtig.

Ein bestimmte, maximale Zahl gleicher Phrasen aufzuzählen ist in BNF doch recht mühsam. Außerdem kann man damit einige Angaben nicht überprüfen, die normalerweise in Programmiersprachen vorkommen:

In Basic besitzen Matrizen normalerweise eine oder zwei Dimensionen. In manchen Versionen von Fortran können Matrizen höchstens sieben Dimensionen besitzen. Ein Compiler muß die Anzahl der Indexausdrücke insgesamt begrenzen, und bei jedem Verweis auf eine Matrix muß kontrolliert werden, daß auch die korrekte, individuelle Anzahl von Indexwerten angegeben wird. In Pascal oder C können beliebig viele Dimensionen verwendet werden; die Anzahl der tatsächlich angegebenen Indexwerte definiert jedoch den Datentyp des Verweises.

Ein ähnliches Problem gibt es für die Parameter von Funktionen. Für die vordefinierten Funktionen sind Anzahl und Typ der Argumente festgelegt. Für die Funktionen, die der Benutzer definiert, ergeben sie sich aus der Definition. Ein Pascal-Compiler muß wenigstens verifizieren, daß Argumentwerte und Parameter zusammenpassen. C ist hier ebenfalls sehr nachsichtig. Ein PL/I-Compiler muß sogar entsprechende Umwandlungen vornehmen.

Die Übergabe von Parametern liefert ein weiteres Problem: wenn, wie in Pascal, ein Unterprogramm ankündigen kann, daß es seine Argumente modifizieren möchte, dann dürfen natürlich auch nur geeignete Argumente übergeben werden. In Fortran können prinzipiell alle Parameter modifiziert werden, aber nur manche Argumente (die *L-Werte* im Sinne von C) ändern sich als Folge einer Zuweisung an einen Parameter - dies ist ein Detailproblem für die Code-Generierung.

Die meisten Semantikprüfungen beziehen sich auf Objekte, die der Benutzer einführt, also auf Konstanten, Typen, Variablen, Unterprogramme und Marken. Wir benötigen eine Symboltabelle, in die alle Information aus den Vereinbarungen eingetragen wird, und die jeweils konsultiert wird, wenn ein vom Benutzer definierter Name verwendet wird.

Einige semantische Einschränkungen behandeln jedoch Probleme, die syntaktisch kompliziert zu formulieren wären:

In Pascal oder C müssen alle Konstanten im Zusammenhang mit **case** voneinander verschieden sein. In C können **case**- und **default**-Marken nur in einer Anweisung vorkommen, wenn diese von **switch** abhängt; diese abhängige Anweisung muß übrigens nicht einmal ein Block sein!

Ganz analog darf eine **break**- oder **continue**-Anweisung in C nur verwendet werden, wenn der entsprechende Sprung auch definiert ist.

Derartige Einschränkungen bedingen gewisse lokale Tests, die sich auf bestimmte syntaktische Konstruktionen beziehen. In unserer *kleinC*-Implementierung werden wir zeigen, wie man zum Beispiel **break** und **continue** mit einem separaten Stack überprüft. Ganz allgemein braucht man hier einen gewissen Einfallsreichtum, denn diese Probleme passen nicht immer in ein vorgefertigtes Schema.

5.2 Symboltabellen

Die Symboltabelle ist die zentrale Ablage für alle Information im Zusammenhang mit Namen und Konstanten, die der Benutzer vereinbart. Der *Inhalt* eines Eintrags in der Symboltabelle hängt von der Information ab, die der Compiler jeweils benötigt, und die aus den Vereinbarungen im Programm bezogen werden kann. Die *Organisation* der Einträge für Suchzwecke reflektiert jedoch die Geltungsbereichsregeln der Programmiersprache:

Basic, zum Beispiel, kann man mit einer Symboltabelle bearbeiten, in die einfach alle neuen Namen aufgenommen werden. Alle Namen sind im Programm global bekannt, folglich muß die Tabelle nur dann bereinigt werden, wenn alle Information über das ganze Programm ausgelöscht wird. Die Parameter einer Benutzerfunktion sind nur innerhalb der Funktion bekannt. Man kann sie daher in eine zweite Tabelle eintragen, die gelöscht wird sobald die Funktion fertig bearbeitet ist.

Für Fortran braucht man im wesentlichen zwei Tabellen: die eine Tabelle enthält Informationen über alle Namen, die in einem Unterprogramm eingeführt werden, und in einer zweiten Tabelle bewahrt man die Unterprogrammnamen selbst auf. Die erste Tabelle kann man im Anschluß an die Übersetzung jedes Unterprogramms wieder löschen. Die zweite Tabelle benötigt man nicht, wenn die Unterprogramme mit einem entsprechenden Lader montiert werden. In diesem Fall gibt man nur die Unterprogrammnamen an den Montierer weiter.

Pascal verlangt streng, daß Namen *vor* Gebrauch vereinbart werden müssen; dazu kommen allerdings verschachtelte Geltungsbereiche. Die Schachtelung der Geltungsbereiche bildet man mit einem Stack als Symboltabelle nach: neue Namen bringt man auf den Stack und am Ende eines Geltungsbereichs, also am Ende einer **function**- oder **procedure**-Definition, wird der Stack wieder entsprechend abgeräumt. Erfolgt dann ein Verweis auf einen Namen, so durchsuchen wir den Stack von der Spitze her und finden auf diese Weise die innerste, aktive Vereinbarung des Namens.

In anderen Sprachen der Algol-Familie müssen Namen nicht unbedingt vor Gebrauch vereinbart werden, solange sich eine Vereinbarung *im Geltungsbereich* des Namens befindet. Dieses Konzept ist einigermaßen kompliziert, denn wir müssen das Programm dann zweimal bearbeiten: im ersten Durchgang muß man alle Vereinbarungen sammeln und jeweils am Anfang ihres Geltungsbereichs zugänglich machen, damit man dann im zweiten Durchgang die Verweise mit dieser Information korrekt verarbeiten kann.

C erlaubt zwar verschachtelte Geltungsbereiche für Variablen, aber Funktionsdefinitionen können nicht verschachtelt sein. Es gibt die Regel 'erst vereinbaren, dann benutzen', mit der Ausnahme daß Funktionen mit **int**

als Resultatwert nicht vereinbart werden müssen. Ein Stack genügt als Symboltabelle, solange wir Funktionsnamen in einer globalen Tabelle aufbewahren, auch wenn sie nur durch implizite lokale Verweise entdeckt werden.

Was speichert man in einem Symboltabelleneintrag? Für Suchzwecke muß der Eintrag Zugriff auf den vom Benutzer verwendeten Namen haben. Zur Überprüfung der Verwendung des Namens müssen wir den Datentyp des Objekts beschreiben. Während der Code-Generierung benötigen wir schließlich Angaben darüber, wie das Objekt repräsentiert wird - Position auf dem Stack, relative oder absolute Adresse, Speicherbedarf, usw. Ist die Symboltabelle als Stack organisiert, müssen wir die Einträge entsprechend verketten; beim Abräumen müssen wir feststellen können, auf welcher Verschachtelungstiefe ein Eintrag definiert wurde.

Den Datentyp eines Objekts zu beschreiben kann schwierig sein. In Fortran gibt es nur wenige Datentypen, zusätzlich kann ein Objekt als Vektor oder Matrix dimensioniert sein; das alles kann man mit ein paar Integer-Werten in der Symboltabelle festhalten. In Sprachen wie Pascal oder C, wo es viele Datentyp-Konstruktoren gibt, muß man oft eine rekursive Beschreibung für einen Datentyp aufbauen, bei der dann normalerweise Zeiger auf weitere Symboltabelleneinträge eine Rolle spielen.

Auch für den Suchvorgang nach einem bestimmten Namen in der Symboltabelle gibt es viele verschiedene Techniken. Hat man bei der lexikalischen Analyse einen Namen oder eine Konstante isoliert, wird man diese normalerweise sofort in der Symboltabelle aufsuchen, oder neu dort eintragen. Von da ab wird immer ein Zeiger in die Symboltabelle weitergegeben, durch den die Information über ein Symbol zugänglich ist, ohne daß ein Name mehr als einmal in der Tabelle aufgesucht werden muß. Die Suche nach dem Namen erfolgt daher in einer Routine, die nur von der lexikalischen Analyse aus aufgerufen wird. Um die Suche zu beschleunigen verwendet man oft zusätzliche Datenstrukturen wie zum Beispiel *hash*-Tabellen. Die Suche benötigt einen Großteil der Rechenzeit bei der lexikalischen Analyse. Es gibt daher sehr viel Literatur über Suchalgorithmen; als Einstieg kann zum Beispiel Kapitel 3.D von W. McKeeman in [Bau76] dienen.

5.3 Beispiel

Für *kleinC* verwenden wir einen Stack als Symboltabelle, den wir als eine linear verkettete Liste von einzelnen Einträgen repräsentieren. Um das Beispiel möglichst einfach zu halten, verwenden wir keine zusätzlichen Datenstrukturen um die Suche zu beschleunigen - dies bleibt als Aufgabe dem Leser überlassen. Die Einträge verwalten wir dynamisch mit Hilfe der **calloc()**- und **cfree()**-Büchereifunktionen.

Den Stack durchsuchen wir von der Spitze her, also von neuen hin zu alten Symbolen. Dadurch entdecken wir die innerste Vereinbarung für einen Namen zuerst, vorausgesetzt, wir entfernen die lokalen Einträge vom Stack, wenn wir einen Block verlassen.

Wir müssen also jeweils wissen, wo die lokalen Vereinbarungen in jedem offenen Block beginnen. Man könnte dies mit einem Stack von Block-Deskriptoren erreichen, von denen aus eine Kette jeweils alle Symboltabelleneinträge in einem Block verbindet. Wir verwenden ein primitiveres Verfahren, das allerdings am Schluß eines Blocks mehr Rechenzeit benötigt: wir verwenden einen globalen Zähler für die Schachtelungstiefe von Blöcken und kopieren den aktuellen Zählerwert jeweils in einen Symboltabelleneintrag, wenn das zugehörige Symbol vereinbart wird. Lokale Einträge auf dem Symboltabellen-Stack sind dann genau diejenigen, die mit der aktuellen Schachtelungstiefe markiert wurden. Bei fehlerhaften Programmen können Symbole vorkommen, die nie vereinbart wurden; diese werden mit einem Initialwert im Feld für Schachtelungstiefe markiert, und diese Symbole werden am Schluß eines Blocks ebenfalls vom Stack entfernt.

Problematisch ist, daß Funktionen vor Gebrauch nicht definiert werden müssen. Da Funktionen jedoch nicht verschachtelt werden können, können wir Symboltabelleneinträge für Funktionen einfach in den äußersten Block verlagern. Unsere Symboltabelle wird dynamisch verkettet, deshalb ist das sehr leicht zu bewerkstelligen: wird ein Symboltabelleneintrag als Funktion vereinbart, so verketten wir ihn einfach am unteren Ende der Tabelle. Wir müssen diese Einträge aufbewahren, da wir einen *load-and-go*-Compiler konstruieren, der die Quelle einmal liest, keinen Montierer verwendet, und der deshalb selbst überprüfen muß, daß auch alle verwendeten Funktionen definiert werden.

Weniger problematisch ist, daß in C die Datentypen von Parametern nicht explizit vereinbart werden müssen: wird ein Name in einer Parameterliste angegeben, so ist er nach Voreinstellung vom Typ **int**. Wir lösen dieses Problem, indem wir die Parameter in der Symboltabelle verketten, wenn sie erstmals in der **parameter_list** auftreten. Wenn die **parameter_declarations** fertig reduziert sind, können wir dieser Kette folgen und die Parameter, falls nötig, als **int** vereinbaren.

Zwar verlangt die Sprachdefinition dies nicht, aber wir wollen doch überprüfen, ob Funktionen konsistent verwendet werden, das heißt, wir wollen wenigstens feststellen, ob sie immer mit der gleichen Anzahl von Argumenten aufgerufen werden.

In einer Hinsicht ist *kleinC* massiv vereinfacht gegenüber C: da es in *kleinC* nur **int**-Werte gibt, haben wir keine Probleme mit unverträglichen Datentypen. Im allgemeinen muß man nämlich in diesem Zusammenhang Semantikprüfungen vornehmen; am besten berechnet man die Resultattypen wenn die Operationen im Parser erkannt werden und reicht entsprechende Beschreibungen auf dem *yacc*-Wert-Stack weiter. Die Lösung ist so umfangreich, daß wir beschlossen haben, das Problem hier dadurch zu umgehen, daß wir keine weiteren Datentypen realisieren.

Als erstes vereinbaren wir einen Symboltabelleneintrag und definieren die möglichen Werte für bestimmte Komponenten. Dies geschieht in der Definitionsdatei *symtab.h*:

```
/*
 *      kleinC -- Definitionsdatei zur Symboltabelle
 */

struct symtab {
        char *  s_name;         /* -> Name */
        int     s_type;         /* Symboltyp */
        int     s_blknum;       /* statisch verschachtelt */
        union {                 /* s.u. */
                int s__num;
                struct symtab * s__link;
                } s__;
        int     s_offset;       /* Symboldefinition */
        struct symtab * s_next; /* -> naechstes Symbol */
        };

#define s_pnum  s__.s__num      /* Anzahl Parameter */
#define NOT_SET (-1)            /* Anzahl unbekannt */
#define s_plist s__.s__link     /* Kette der Parameter */

/*
 *      Werte fuer s_type
 */

#define UDEC    0       /* nicht vereinbart */
#define FUNC    1       /* Funktion */
#define UFUNC   2       /* undefinierte Funktion */
#define VAR     3       /* definierte Variable */
#define PARM    4       /* undeklarierter Parameter */

/*
 *      s_type Darstellung fuer S_TRACE
 */

#define SYMmap  "nicht vereinbart", "Funktion", \
                "undefinierte Funktion", "Variable", \
                "Parameter"

/*
 *      Typvereinbarungen fuer Funktionen der Symboltabelle
 */

struct symtab * link_parm();  /* Parameter verketten */
struct symtab * s_find();     /* Symbol nach Namen aufsuchen */
struct symtab * make_parm();  /* Parameter deklarieren */
struct symtab * make_var();   /* Variable definieren */
struct symtab * make_func();  /* Funktion definieren */
```

```
/*
 *      Typvereinbarungen fuer Buechereifunktionen
 */

char * strsave();       /* Zeichenkette dynamisch speichern */
char * calloc();        /* dynamisch Speicher beschaffen */
```

s_name verwenden wir bei der Suche nach einem Namen; dabei verwenden wir **strsave()** um jeden neuen Namen als Zeichenkette dynamisch zu speichern. **s_blknum** enthält die Schachtelungstiefe des Geltungsbereichs, in dem der Name vereinbart wurde. **s_offset** beschreibt später den Zugriff zur Laufzeit auf das Objekt; momentan ist diese Komponente überflüssig. **s_next** verkettet die Symboltabelleneinträge als Stack.

s__ hat eine doppelte Aufgabe: durch die erste Alternative **s__link** (zur leichteren Verwendung auch als **s_plist** definiert) verläuft die Kette, die alle Parameter einer Funktion verbindet, und mit deren Hilfe wir dann undeklarierte Parameter leicht vereinbaren können; bei Funktionsnamen wird mit der zweiten Alternative **s__num** (alias **s_pnum**) die konsistente Verwendung kontrolliert. Die zweite Alternative hat zuerst den Wert **NOT_SET**. Wird die Funktion zum ersten Mal definiert oder verwendet (dies ist der Fall, wenn der Wert noch **NOT_SET** ist), erhält **s_pnum** als Wert die Anzahl der Argumente oder Parameter, die an diesem Punkt spezifiziert wurden. Bei jeder weiteren Verwendung der Funktion (also wenn der Wert dann von **NOT_SET** verschieden ist), kann dann diese Zahl kontrolliert werden.

s_type schließlich repräsentiert den Datentyp des Symboltabelleneintrags. Diese Komponente ist eigentlich zu primitiv - sobald Zeiger und Vektoren (ganz zu schweigen von Strukturen) zur Sprachdefinition hinzukommen, muß eine wesentlich aufwendigere Beschreibung verwendet werden. Im Moment benötigen wir sehr wenig: ein Name wird zunächst als 'nicht vereinbart' eingetragen (**UDEC**); wird der Name als der einer Funktion verwendet aber noch nicht definiert, erhält **s_type** den Wert **UFUNC**; wird die Funktion definiert, ändern wir den Wert in **FUNC**; alternativ kann ein Name einen Parameter (**PARM**) oder eine Variable (**VAR**) bezeichnen, bei beiden kann es sich nur um **int**-Werte handeln.

Zur Ablaufverfolgung geben wir die Symboltabelleneinträge aus. Um **s_type** nicht nur numerisch ausgeben zu müssen, unterhalten wir **SYMmap** parallel zur numerischen Repräsentierung. In bezug auf die leichte Modifizierbarkeit der Symboltabellenroutinen ist diese Codierung zwar etwas riskant, aber wir haben die zwei Darstellungen wenigstens nahe beieinander definiert.

Typischer Programmierstil ist übrigens auch, daß wir alle Komponentennamen eines Symboltabelleneintrags mit **s_** begonnen haben.

Wie beim Formatierer sammeln wir auch alle Symboltabellenroutinen in einer eigenen Datei *symtab.c*. Zur Grammatik fügen wir jetzt Aktionen hinzu, die diese Funktionen für Semantikprüfungen aufrufen. Bei *kleinC* muß im wesentlichen jedes Auftreten von **Identifier** überprüft werden. **break** und **continue** sollten wir auch

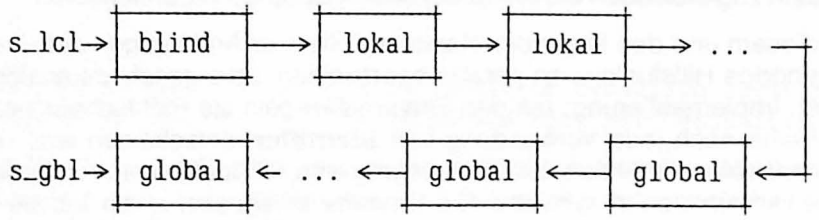
kontrollieren - dieses Problem sparen wir uns aber für die Code-Generierung auf, wo wir dann zugleich auch die entsprechenden Sprünge implementieren.

In diesem und den folgenden Kapiteln wollen wir Änderungen der Grammatik und zugehörige Hilfsfunktionen parallel beschreiben. Dies geschieht analog zur eigentlichen Implementierung: mit den Grammatikregeln als Richtschnur betrachten wir der Reihe nach jede Verwendung von **Identifier**, entscheiden was überprüft werden muß, rufen als Aktion der Formulierung eine Hilfsfunktion auf, und implementieren die Hilfsfunktion in *symtab.c*. Die Grammatik liegt explizit als Tabelle vor, dadurch sind wir praktisch gezwungen auch alle Fälle zu untersuchen. Die Grammatik ist eine ideale Kontrollstruktur, gerade auch zur Strukturierung des Implementierungsvorgangs. Im Abschnitt 4 im Anhang können alle Hilfsfunktionen für die Symboltabellenverwaltung im Zusammenhang studiert werden.

Wir beginnen mit der Definition der notwendigen globalen Variablen. Da sie exklusiv zur Symboltabellenverwaltung verwendet werden, definieren wir sie **static**. Auf diese Weise verbirgt *symtab.c* seine privaten Daten vor den anderen Modulen des Compilers. Analog exportieren wir aus diesem Modul - indem wir bei einem globalen Namen das **static**-Attribut *nicht* angeben - nur die Funktionen, die auch wirklich von Aktionen im Parser-Modul aus aufgerufen werden. Alle internen Hilfsfunktionen werden als **static**-Objekte verborgen.

```
/*
 *      kleinC -- Symboltabellen-Manager
 */

#include "symtab.h"
#include "y.tab.h"

/*
 *      Symboltabelle
 */

static  struct  symtab
        symtab,                 /* blindes Element */
        * s_gbl;                /* Ende der globalen Kette */
#define s_lcl   (& symtab)      /* Ende der lokalen Kette */

/*
 *      Blocktabelle
 */

static int blknum = 0;          /* Verschachtelungstiefe */
```

Die Symboltabelle hat folgende, dynamisch aufgebaute Struktur:



s_lcl zeigt auf ein blindes Element, das sich nie ändert und nach dem neue Einträge hinzugefügt werden. Entlang der Kette, am Stack-Ende, sind die globalen Einträge, die erst ganz zum Schluß entfernt werden. Die Funktion **s_create()** fügt einen neuen, lokalen Eintrag zur Symboltabelle hinzu, mit dem Namen, der als Argument übergeben wird. Der neue Eintrag wird als noch nicht vereinbart markiert.

```
static struct symtab * s_create(name)
        register char * name;
{       register struct symtab * new_entry = (struct symtab *)
                        calloc(1, sizeof(struct symtab));

        if (new_entry)
        {       new_entry->s_next = s_lcl->s_next;
                s_lcl->s_next = new_entry;
                new_entry->s_name = strsave(name);
                new_entry->s_type = UDEC;
                new_entry->s_blknum = 0;
                new_entry->s_pnum = NOT_SET;
                return new_entry;
        }
        fatal("Kein Platz mehr fuer Symbole.");
        /*NOTREACHED*/
}
```

s_gbl zeigt auf den *letzten* (untersten) globalen Eintrag. Wird ein Eintrag für eine Funktion an das globale Ende der Kette verlegt, so wird er so verkettet, daß er auf das Element bei **s_gbl** folgt. Anschließend lassen wir dann **s_gbl** auf diesen neuen, globalen Eintrag zeigen. Dies erledigt die Funktion **s_move()** die als Argument einen Zeiger auf den Eintrag erhält, der umgehängt werden soll:

```
static s_move(symbol)
        register struct symtab * symbol;
{       register struct symtab * ptr;

        /* auf der Kette suchen (muss vorhanden sein!) */
        for (ptr=s_lcl; ptr->s_next != symbol; ptr=ptr->s_next)
                if (! ptr->s_next)
                        bug("s_move");
```

```
        /* ausketten */
        ptr->s_next = symbol->s_next;

        /* am globalen Ende einhaengen */
        s_gbl->s_next = symbol;
        s_gbl = symbol;
        s_gbl->s_next = (struct symtab *) 0;
}
```

Man beachte, daß der Eintrag nur in eine neue Position im Symboltabellen-Stack eingehängt wird. Im Speicher wird der Eintrag dabei nicht verlagert, so daß sich der Zeigerwert, der auf den Eintrag verweist und der als Argument übergeben wird, selbst nicht ändert.

An der Stack-Spitze haben wir ein blindes Element angelegt, so daß wir nicht überprüfen müssen, ob wir gerade die aktuelle Stack-Spitze verlagern, das heißt, daß wir in keinem Fall **s_lcl** verändern müssen.

Am Anfang darf allerdings **s_gbl** nicht auf das blinde Element zeigen. Wäre das der Fall, würden wir den ersten (globalen) Eintrag *im Anschluß* an **s_gbl** einfügen. Lokale Einträge könnten dann *zwischen* **s_gbl** und diesen globalen Eintrag rutschen; irgendwann wird ein solcher Eintrag verlagert, womit er global wird... und totales Chaos wäre die Folge! **s_gbl** muß so initialisiert werden, daß dieser Zeiger auf einen globalen Eintrag zeigt - ein blindes Element an dieser Stelle würde die Symboltabelle teilen und so einen weiteren Sonderfall bilden. Erfreulicherweise gibt es aber sogar einen nützlichen globalen Eintrag: jedes *kleinC*-Programm muß eine **main()**-Funktion beinhalten. Zur Initialisierung eröffnen wir daher den äußersten Block und initialisieren **s_gbl** als Zeiger auf einen Eintrag für **main** als Namen einer undefinierten Funktion:

```
init()
{
        blk_push();
        s_gbl = s_create("main");
        s_gbl->s_type = UFUNC;
}
```

init() muß aufgerufen werden, bevor auf die Symboltabelle zugegriffen werden kann. **init()** könnte man daher aus der **main()**-Funktion unseres Compilers heraus aufrufen, vor dem Aufruf von **yyparse()**. Sichtbarer ist die Lösung, bei der wir **init()** sehr früh vom Parser selbst aus aufrufen. Dies ist dann die erste Aktion, die wir zum Parser hinzufügen:

```
program
        :       { init(); }
          definitions
                { blk_pop(); }
```

So wird **init()** vor allen Aufrufen der lexikalischen Analyse aufgerufen.

init() eröffnet auch den äußersten Block:

```
blk_push()
{
        ++ blknum;
}
```

Zu jedem Aufruf von **blk_push()** gehört ein entsprechender Aufruf von **blk_pop()**, damit die anschließend unzugänglichen Symbole wieder vom Symboltabellen-Stack entfernt werden, und damit undefinierte Funktionen entdeckt werden, etc. Wir verschieben jedoch eine Diskussion von **blk_pop()**, bis wir gesehen haben, wie Symbole wirklich in die Symboltabelle eingetragen werden.

Jeder vom Benutzer eingeführte Name taucht zuerst in der lexikalischen Analyse auf. **yylex()** muß jedes Symbol in die Symboltabelle eintragen, wenn es dort nicht schon vorhanden ist. Zu diesem Zweck haben wir bereits Aufrufe einer Funktion **s_lookup()** in die lexikalische Analyse eingebracht. **s_lookup()** wird aufgerufen, wenn sich das Eingabesymbol in **yytext[]** befindet und erhält den gewünschten Symbolwert (**Constant** oder **Identifier**) als Argument. Bei *kleinC* genügt es, wenn man den Text einer **Constant** dynamisch speichert. Ein **Identifier** muß in der Symboltabelle aufgesucht oder eingetragen werden:

```
s_lookup(yylex)
        int yylex;              /* Constant oder Identifier */
{       extern char yytext[];   /* Darstellung des Symbols */

        switch (yylex) {
        case Constant:
                yylval.y_str = strsave(yytext);
                break;
        case Identifier:
                if (yylval.y_sym = s_find(yytext))
                        break;
                yylval.y_sym = s_create(yytext);
                break;
        default:
                bug("s_lookup");
        }
}
```

In jedem Fall enthält **yylval** schließlich einen Zeiger, mit dessen Hilfe auf die semantische Information für das vom Benutzer eingeführte Eingabesymbol zugegriffen werden kann. Wir wissen zwar, daß **s_lookup()** nur mit **Constant** oder **Integer** als Argumentwert aufgerufen wird; wir überprüfen den Argumentwert aber trotzdem im Interesse eines robusten Programms.

Wir haben bisher das Problem vertagt, wie wir einen Eintrag nach Namen in der Symboltabelle suchen. Dies ist eine der weniger effizienten Funktionen in dieser Implementierung:

```
struct symtab * s_find(name)
        char * name;
{       register struct symtab * ptr;

        /* Symboltabelle absuchen */
        for (ptr = s_lcl->s_next; ptr; ptr = ptr->s_next)
                if (! ptr->s_name)
                        bug("s_find");
                else
                        /* bei gleichen Namen: -> Eintrag */
                        if (strcmp(ptr->s_name, name) == 0)
                                return ptr;
        /* Suche erfolglos: NULL */
        return (struct symtab *) 0;
}
```

Mit **s_lookup()** hat **yylex()** dafür gesorgt, daß für jeden **Identifier** ein Eintrag in der Symboltabelle vorhanden ist. Ein neuer Eintrag ist noch **UDEC**, aber falls der Name schon früher auftrat, muß **yylval** nicht unbedingt auf den richtigen Eintrag zeigen. Wir müssen die Dinge überall dort reparieren oder überprüfen, wo **Identifier** in der Grammatik auftaucht.

Da Namen in C - mit Ausnahme von Funktionsnamen - vereinbart werden müssen, bevor sie benutzt werden können, werden Vereinbarungen zuerst reduziert. Betrachten wir daher den Punkt, an dem ein **Identifier** in einer **parameter_list** auftritt, das heißt, wo ein **Identifier** als neuer Parametername vereinbart wird:

```
parameter_list
        : Identifier
                { $$ = link_parm($1, (struct symtab *) 0); }
        | parameter_list ',' Identifier
                { $$ = link_parm($3, $1);
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | parameter_list error
        | parameter_list error Identifier
                { $$ = link_parm($3, $1);
                  yyerrok;
                }
        | parameter_list ',' error
```

link_parm() und der *yacc*-Wert-Stack werden benutzt um alle Parameternamen in der **parameter_list** zu verketten. **link_parm()** muß auch den **Identifier** in der Symboltabelle als **PARM** markieren. Später verwenden wir dann die Parameterkette, um den Typ von nicht deklarierten Parametern festzulegen, und um die Anzahl der Parameter festzustellen. Betrachten wir zuerst **link_parm()**:

```
struct symtab * link_parm(symbol, next)
        register struct symtab * symbol, * next;
{
        switch (symbol->s_type) {
        case PARM:
                error("Parameter %s doppelt", symbol->s_name);
                return next;
        case FUNC:
        case UFUNC:
        case VAR:
                symbol = s_create(symbol->s_name);
        case UDEC:
                break;
        default:
                bug("link_parm");
        }
        symbol->s_type = PARM;
        symbol->s_blknum = blknum;
        symbol->s_plist = next;
        return symbol;
}
```

yylex() hat den **Identifier** schon in der Symboltabelle aufgefunden; der dabei konstruierte Zeigerwert wird an **link_parm()** übergeben. Die **s_type**-Komponente dieses Eintrags bestimmt, was getan werden muß:

Ist der Eintrag noch **UDEC**, so handelt es sich um einen neuen Namen, der selbst als Parameter vereinbart werden kann.

Hat der Eintrag schon einen Typ, ist aber kein **PARM**, so beschreibt der Eintrag einen globalen **Identifier**, der gerade durch einen Parameter verborgen wird. Wir müssen deshalb einen neuen, lokalen Symboltabelleneintrag für den gleichen Namen mit **s_create()** anlegen, der dabei automatisch oben auf den Stack gebracht wird und so den globalen Eintrag für den gleichen Namen verbirgt. Der lokale Eintrag kann dann als **PARM** vereinbart werden.

Ist der Eintrag schon ein **PARM**, müssen wir uns darüber beschweren, daß der gleiche Name zweimal als Parameter in der gleichen Parameterliste verwendet wird.

Der Entwurf dieser **link_parm()**-Funktion war relativ leicht: die Grammatik isoliert den Kontext von diesem Teil der Semantikprüfung, und die möglichen Werte der **s_type**-Komponente im Symboltabelleneintrag sorgen dafür, daß wir alle möglichen Fälle betrachten. Da in C Funktionen nicht verschachtelt werden können, ist die Behandlung dieser Situation einfach.

Einen Parameter zu deklarieren, ist etwas schwieriger. Dies passiert, wenn ein **Identifier** in einer **parameter_declarator_list** vorkommt:

```
parameter_declarator_list
        : Identifier
                { make_parm($1); }
        | parameter_declarator_list ',' Identifier
                { make_parm($3);
                  yyerrok;
                }
        | error
        | parameter_declarator_list error
        | parameter_declarator_list error Identifier
                { make_parm($3);
                  yyerrok;
                }
        | parameter_declarator_list ',' error
```

make_parm() muß **PARM** in der Symboltabelle in **VAR** abändern und dadurch den Parameter deklarieren. Sind alle **parameter_declarations** erledigt, müssen wir alle restlichen **PARM**-Einträge als Integer-Variablen deklarieren, das heißt, ebenfalls als **VAR**.

make_parm() erhält einen Symboltabelleneintrag, den **yylex()** mit Hilfe von **s_lookup()** aufgefunden oder erzeugt hat. Wenn wir wieder alle möglichen Werte für **s_type** in diesem Eintrag betrachten, sehen wir eine Reihe von Fehlermöglichkeiten:

Ist der Eintrag bereits als **VAR** markiert, und zwar mit Verschachtelungstiefe zwei, dann haben wir zwei Deklarationen für den gleichen Parameter. Dies ist natürlich anders als die zweimalige Verwendung des gleichen Namens in einer **parameter_list**.

Taucht **VAR** mit einer anderen (hoffentlich globalen!) Schachtelungstiefe auf, oder finden wir in **s_type** noch einen anderen Wert als **PARM**, müssen wir als Fehler vermerken, daß ein Name als Parameter vereinbart werden soll, der in der **parameter_list** nicht vorkam. Aus Sicherheitsgründen erzeugen wir in allen diesen Fällen einen neuen, doppelten Eintrag für den Namen und vereinbaren ihn dann als **VAR**.

```
struct symtab * make_parm(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case VAR:
                if (symbol->s_blknum == 2)
                {       error("Parameter %s zweimal deklariert",
                                symbol->s_name);
                        return symbol;
                }
        case UDEC:
        case FUNC:
        case UFUNC:
                error("%s ist kein Parameter", symbol->s_name);
                symbol = s_create(symbol->s_name);
        case PARM:
                break;
        default:
                bug("make_parm");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
        return symbol;
}
```

Als nächstes definieren wir eine Variable. Dies passiert im gleichen Grammatik-Kontext wie die Deklaration von Parametern. Die nötigen semantischen Aktionen sind jedoch ganz anders. Wir haben deshalb separate Grammatikbegriffe für Variablendefinitionen und Parameterdeklarationen eingeführt. Wenn wir *kleinC* in Richtung auf C erweitern, müssen wir diesen Unterschied ohnehin machen: Variablen kann man initialisieren, Parameter nicht.

```
declarator_list
        : Identifier
                { make_var($1); }
        | declarator_list ',' Identifier
                { make_var($3);
                  yyerrok;
                }
        | error
        | declarator_list error
        | declarator_list error Identifier
                { make_var($3);
                  yyerrok;
                }
        | declarator_list ',' error
```

make_var() muß die Definition in der Symboltabelle festhalten. Als Argument übergeben wir wieder den Symboltabelleneintrag, den **yylex()** gefunden oder erzeugt hat. Auch dieser Eintrag hat schon einen Wert in der **s_type**-Komponente, und wir betrachten wieder die einzelnen Fälle:

Wurde der Eintrag noch nicht vereinbart, haben wir einen neuen Namen vor uns, den wir als Variable im aktuellen Block vereinbaren können.

Handelt es sich bei dem Eintrag um eine Funktion oder eine Variable im gleichen Block, melden wir, daß ein Name zweimal vereinbart wird. Andernfalls müssen wir den Symboltabelleneintrag mit **s_create()** kopieren, damit wir einen neuen, lokalen Eintrag erhalten, den wir dann wie gewünscht definieren können.

Zusätzlich müssen wir den Fall ausschließen, daß eine lokale Variable den gleichen Namen erhält wie ein Parameter. Der Parameter ist zu diesem Zeitpunkt auch als **VAR** markiert, aber wie wir weiter unten sehen werden, befindet er sich auf Schachteltiefe zwei, und die lokale Variable hat (wenigstens) Schachteltiefe drei. Dieser spezielle Fall kann bei Funktionen natürlich nicht auftreten.

Schließlich sollten wir den Wert **PARM** in **s_type** nicht antreffen können, da alle derartigen Einträge schon entweder deklariert oder implizit vereinbart worden sein sollten. Um uns gegen üble Eingabefehler abzusichern, geben wir auch in diesem Fall eine Fehlermeldung aus.

```
struct symtab * make_var(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case VAR:
        case FUNC:
        case UFUNC:
                if (symbol->s_blknum == blknum
                    || symbol->s_blknum == 2 && blknum == 3)
                        error("Name %s zweimal vereinbart",
                                symbol->s_name);
                symbol = s_create(symbol->s_name);
        case UDEC:
                break;
        case PARM:
                error("Parameter %s nicht vorgesehen",
                        symbol->s_name);
                break;
        default:
                bug("make_var");
        }
```

```
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
        return symbol;
    }
```

Die vorstehende Funktion befaßt sich mit den meisten Problemen der Blockstruktur. Trotzdem ergibt sich alles, was passieren kann, aus den möglichen Werten für **s_type** und den Schachteltiefen, bei denen wir diese Werte bei einem früher vereinbarten Symbol entdecken.

Wir kennen schon einiges, was wir tun müssen um eine Funktion zu definieren: die Länge der **parameter_list** muß berechnet und in die Symboltabelle eingetragen oder mit dem dort vorhandenen Wert verglichen werden, und wir müssen die Schachteltiefe für Parameter korrekt gestalten. Letzteres erledigen wir, indem wir die Parametermanipulationen mit zwei zusätzlichen Aufrufen von **blk_push()** und **blk_pop()** umgeben. Betrachten wir die Definition einer Funktion in der Grammatik:

```
function_definition
        : Identifier '('
                { make_func($1); blk_push(); }
          optional_parameter_list rp
          declarations
                { chk_parm($1, parm_default($4)); }
          compound_statement
                { blk_pop(); }

optional_parameter_list
        : /* keine Parameter */
                { $$ = (struct symtab *) 0; }
        | parameter_list
                /* $$ = $1 = Kette der Parameter */
```

make_func() muß die Funktion in der Symboltabelle definieren. **chk_parm()** erhält einen Zeiger auf den Symboltabelleneintrag für die Funktion und die Anzahl der angegebenen Parameter. Die Funktion muß diese Anzahl überprüfen, oder beim ersten Mal in die Symboltabelle eintragen. **parm_default()** bearbeitet alle Parameter, die in der **parameter_list** vorkamen, aber in den **parameter_declarations** nicht besonders vereinbart wurden. In C werden diese Parameter dann implizit als **int**-Variable vereinbart; die **s_type**-Komponente muß den Wert **PARM** haben, und wir ändern ihn ab in **VAR**. **parm_default()** muß außerdem die Parameter zählen und ihre Anzahl als Resultat liefern. Betrachten wir die Funktionen der Reihe nach:

```
struct symtab * make_func(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UFUNC:
        case UDEC:
                break;
        case VAR:
                error("Funktion %s dupliziert globale Variable",
                        symbol->s_name);
                return symbol;
        case FUNC:
                error("Function %s zweimal definiert",
                        symbol->s_name);
                return symbol;
        default:
                bug("make_func");
        }
        symbol->s_type = FUNC;
        symbol->s_blknum = 1;
        return symbol;
}
```

Wenn **make_func()** aufgerufen wird, sind wir im globalen Bereich und brauchen deshalb den Symboltabelleneintrag nicht mit **s_move()** zu verlagern. Ein **UDEC**- oder **UFUNC**-Eintrag kann einfach als Funktion definiert werden, ein **VAR**- oder **FUNC**-Eintrag wird gerade doppelt definiert und ein anderer Wert kann für **s_type** legitimerweise nicht auftreten.

Im Moment benötigen wir keine Resultatwerte von **make_var()**, **make_parm()** und **make_func()**. Wenn wir aber zur Code-Generierung kommen, werden wir die korrigierten Symboltabellenzeiger benötigen, damit wir zusätzliche Informationen eintragen können, wie etwa die Adresse einer Variablen oder einer Funktion. Die Funktionen liefern deshalb schon jetzt einen entsprechenden Resultatwert. Wenn etwas schiefgeht, wenn wir also zum Beispiel eine doppelte Definition finden, kann man nicht mehr viel retten. Ein Compiler muß kein Hellseher sein, er darf nur nicht kaputtgehen, auch nicht bei den rücksichtslosesten Versuchen des Benutzers!

Die Anzahl der Parameter einzutragen oder zu prüfen ist sehr leicht:

```
chk_parm(symbol, count)
        register struct symtab * symbol;
        register int count;
{
        if (symbol->s_pnum == NOT_SET)
                symbol->s_pnum = count;
        else if ((int) symbol->s_pnum != count)
             warning("Funktion %s hat eigentlich %d Argument(e)",
                        symbol->s_name, symbol->s_pnum);
}
```

Nicht vereinbarte Parameter zu vereinbaren ist genauso leicht. Wir müssen nur der Kette durch die Symboltabelle folgen:

```
int parm_default(symbol)
        register struct symtab * symbol;
{       register int count = 0;

        while (symbol)
        {       ++ count;
                if (symbol->s_type == PARM)
                        symbol->s_type = VAR;
                symbol = symbol->s_plist;
        }
        return count;
}
```

Fast schon das große Finale: nachdem jetzt alle Symbole in die Symboltabelle eingetragen wurden, untersuchen wir jetzt, wie wir sie am Schluß eines Blocks wieder eliminieren. **blk_pop()** wird für den globalen Bereich von einer Aktion bei **program** aus aufgerufen, für den Parameter-Block von einer Aktion, die zur Funktionsdefinition gehört, und schließlich auch von einer Aktion im Zusammenhang mit **compound_statement**:

```
compound_statement
        : '{'
                { blk_push(); }
          declarations statements rr
                { blk_pop(); }
```

Wir brauchen eine Schleife, die die momentane Schachteltiefe in der Symboltabelle durchläuft, damit wir alle lokalen Einträge finden und freigeben können. Wenn wir undefinierte Einträge finden, haben wir normalerweise bereits entsprechende Fehlermeldungen ausgegeben; undefinierte Funktionen finden wir jedoch genau dann, wenn wir den globalen Block abschließen.

```
blk_pop()
{       register struct symtab * ptr;

        for (ptr = s_lcl->s_next;
             ptr &&
             (ptr->s_blknum >= blknum || ptr->s_blknum == 0);
             ptr = s_lcl->s_next)
        {
                if (! ptr->s_name)
                        bug("blk_pop: Name ist NULL");
#ifdef TRACE
        {       static char * type[] = { SYMmap };

                message(
                    "Entfernt wird %s: %s, Tiefe %d, Offset %d",
                        ptr->s_name, type[ptr->s_type],
                        ptr->s_blknum, ptr->s_offset);
        }
#endif TRACE
                if (ptr->s_type == UFUNC)
                        error("undefinierte Funktion %s",
                                ptr->s_name);
                cfree(ptr->s_name);
                s_lcl->s_next = ptr->s_next;
                cfree(ptr);
        }
        -- blknum;
}
```

Das blinde Element am Anfang der Kette müssen wir ignorieren, wir dürfen einem Nullzeiger nicht aus der Symboltabelle folgen (eigentlich sollten wir einen derartigen Zeiger gar nicht finden) und wir müssen alle Einträge mit der aktuellen Schachteltiefe bearbeiten, die hoffentlich **blknum** ist! Wir können auch ein paar Einträge mit Schachteltiefe null finden - also undefinierte Einträge. Diese ergeben sich zum Beispiel, wenn Eingabesymbole während der Fehlerbehandlung ignoriert werden. Ja - wir können Symbole finden, die unsere *reduce*-Aktionen nie gesehen haben!

blk_pop() eignet sich gut zum Einbau einer Ablaufverfolgung, damit wir sehen, was unser Compiler eigentlich tut. Wir fügen, unter Kontrolle von **#ifdef**-Preprozessor-Anweisungen, Code ein, damit alle Komponenten eines Symboltabelleneintrags gezeigt werden, bevor er freigegeben wird.

Bisher haben wir uns sehr darum bemüht, Information in die Symboltabelle einzutragen. Jetzt müssen wir diese Information auch benutzen. Immer wenn **Identifier** in einer Formulierung verwendet wird, müssen wir den Symboltabelleneintrag unter-

suchen, ob die Verwendung in diesem Kontext auch erlaubt ist. In *kleinC* gibt es dabei im wesentlichen nur zwei Situationen: Die eine ist

```
binary
        : Identifier
                { chk_var($1); }
        | PP Identifier
                { chk_var($2); }
        | Identifier '=' binary
                { chk_var($1); }
```

(Wir haben eine Reihe von ähnlichen Formulierungen hier ausgelassen.)

chk_var() erhält den Symboltabellenzeiger, der zu einem **Identifier** gehört, und muß untersuchen, ob dieser **Identifier** auf beiden Seiten einer Zuweisung als Variable verwendet werden kann. Die verschiedenen Werte von **s_type** bestimmen auch hier das Resultat:

```
chk_var(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UDEC:
                error("undefinierte Variable %s",
                        symbol->s_name);
                break;
        case PARM:
                error("Parameter %s unerwartet",
                        symbol->s_name);
                break;
        case FUNC:
        case UFUNC:
                error("Funktion %s als Variable verwendet",
                        symbol->s_name);
        case VAR:
                return;
        default:
                bug("check_var");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
}
```

Wir schützen uns hier sorgfältig gegen eine Kaskade von Fehlermeldungen über die gleiche undefinierte Variable, oder über den gleichen Parameter, den wir vermissen. Finden wir ein derartiges Objekt, geben wir eine einzige Fehlermeldung aus und definieren das Objekt dann im aktuellen Block. Auf diese Weise vermeiden wir weitere

Fehlermeldungen. Dies erklärt auch, warum eine Schleife über die aktuelle Schachteltiefe in der Symboltabelle Einträge mit größerer Schachteltiefe überhaupt finden kann: sie resultieren von dieser Reparatur.

Der zweite Test bezieht sich auf die Verwendung von **Identifier** als Funktionsname in einem Funktionsaufruf. Hier überprüfen wir auch die Anzahl Argumente mit der früher definierten Funktion **chk_parm()**. Wir müssen jedoch noch die Argumente zählen:

```
binary
        : Identifier '('
                { chk_func($1); }
          optional_argument_list rp
                { chk_parm($1,$4); }

optional_argument_list
        : /* keine Argumente */
                { $$ = 0; }
        | argument_list
                /* $$ = $1 = Anzahl Argumente */

argument_list
        : binary
                { $$ = 1; }
        | argument_list ',' binary
                { ++ $$;
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | argument_list error
        | argument_list ',' error
```

Hier ist **chk_func()**, die letzte Funktion im Symboltabellenmodul:

```
chk_func(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UDEC:
                break;
        case PARM:
                error("Parameter %s unerwartet",
                        symbol->s_name);
                symbol->s_pnum = NOT_SET;
                return;
```

```
                case VAR:
                        error("Variable %s als Funktion verwendet",
                                symbol->s_name);
                        symbol->s_pnum = NOT_SET;
                case UFUNC:
                case FUNC:
                        return;
                default:
                        bug("check_func");
                }
                s_move(symbol);
                symbol->s_type = UFUNC;
                symbol->s_blknum = 1;
        }
```

Finden wir einen neuen Namen als Funktionsnamen bei einem Aufruf, wird dieser Name implizit als Funktionsname vereinbart. Dazu müssen wir seinen Eintrag in der Symboltabelle mit Hilfe von **s_move()** verlagern und als undefinierte, globale Funktion markieren.

Gab es davor keine Fehler in der Eingabe, sollten wir jetzt keinen Parameter entdecken können. Falls wir doch einen entdecken, sorgen wir dafür, daß ein späterer Aufruf von **chk_parm()** ohne Fehlermeldung verläuft: wir geben vor, daß die Anzahl der Parameter für die Funktion noch **NOT_SET** ist. Gegen Variable, die als Funktionen mißbraucht werden, schützen wir uns ähnlich.

Damit ist die Symboltabelle vollständig implementiert. Wir haben stillschweigend angenommen, daß bestimmte Funktionen existieren, mit denen wir verschiedene Fehlermeldungen und Warnungen ausgeben können:

message()

gibt eine einfache Meldung aus mit einer von **yywhere()** produzierten Positionsangabe (vergleiche Abschnitt 3.3).

warning()

gibt eine Meldung aus, die als Warnung deklariert wird.

error()

gibt eine Meldung aus, die als Fehler deklariert und gezählt wird.

fatal()

gibt eine Meldung aus, deklariert sie als bösen Fehler, und bricht den Prozeß ab. Diese Funktion wird aufgerufen, wenn dem Compiler Betriebsmittel wie Speicher ausgehen, oder wenn er sonst die Übersetzung nicht vollständig vornehmen kann.

bug()

sollte 'nie' passieren. Eine Meldung wird ausgegeben, als Systemfehler deklariert, und der Prozeß wird abgebrochen. Diese Funktion wird nur in Situationen aufgerufen, die nicht eintreten können wenn der Compiler selbst korrekt ist. Wir ziehen es vor, hier nur eine kurze Meldung auszugeben - tritt der Systemfehler wirklich ein, würden wir wohl den Compiler mit einer umfangreicheren Fehlermeldung nochmals übersetzen, um zu sehen, was wirklich passiert ist. Dadurch reduzieren wir zwar die Größe des Produktionscompilers, aber in wirklich bösen Fällen reproduziert der veränderte Compiler den Fehler möglicherweise nicht mehr.

Alle diese Funktionen werden genau wie die Büchereifunktion **printf()** aufgerufen, also mit einem Format und einer kleinen, variablen Anzahl von Werten, die ausgegeben werden sollen. Die Funktionen selbst befinden sich im Abschnitt 3 im Anhang.

Das folgende Programm enthält absichtlich eine Reihe von Fehlern:

```
/*
 *      Symboltabellen-Demonstration, mit Fehlerbehandlung
 */

main(a,b) int a,b;
{       a=b;
        {a;b;}
        if (a==b) {a;b;}
        if (a==b+1) a; else b;
        while (a==b) {a; break;}
        return;
}
int f() { int x; int y; return x+y; }
f(a,b)                  /* 14: Funktion zweimal definiert */
        int a;          /* undeklarierter Parameter b (kein Fehler) */
{       int b,c;        /* 16: Name b dupliziert Parameter b */
        int c,d;        /* 17: Name c zweimal vereinbart */
        e=b;            /* 18: undefinierte Variable e */
        f=c+d;          /* 19: Funktion als Variable verwendet */
        a(b);           /* 20: Variable als Funktion verwendet */
        g();            /* 21: undefinierte Funktion g */
}
h(a,a)                  /* 23: Parameter doppelt */
        int f;          /* 24: kein Parameter */
        int a;
        int a;          /* 26: Parameter zweimal */
{ }
```

Syntax- und Semantikanalyse dieses Programms liefern folgende Ausgabe (wenn **TRACE** definiert ist):

```
Zeile 12 bei "}": Entfernt wird b: Variable, Tiefe 2, Offset 0
Zeile 12 bei "}": Entfernt wird a: Variable, Tiefe 2, Offset 0
Zeile 13 bei "}": Entfernt wird y: Variable, Tiefe 3, Offset 0
Zeile 13 bei "}": Entfernt wird x: Variable, Tiefe 3, Offset 0
[Fehler 1] Zeile 14 bei "(": Function f zweimal definiert
[Warnung] Zeile 16 bei "{": Funktion f hat eigentlich 0 Argument(e)
[Fehler 2] Zeile 16 bei "b": Name b zweimal vereinbart
[Fehler 3] Zeile 17 bei "c": Name c zweimal vereinbart
[Fehler 4] Zeile 18 bei ";": undefinierte Variable e
[Fehler 5] Zeile 19 bei ";": Funktion f als Variable verwendet
[Fehler 6] Zeile 20 bei "(": Variable a als Funktion verwendet
Zeile 22 bei "}": Entfernt wird e: Variable, Tiefe 3, Offset 0
Zeile 22 bei "}": Entfernt wird d: Variable, Tiefe 3, Offset 0
Zeile 22 bei "}": Entfernt wird c: Variable, Tiefe 3, Offset 0
Zeile 22 bei "}": Entfernt wird c: Variable, Tiefe 3, Offset 0
Zeile 22 bei "}": Entfernt wird b: Variable, Tiefe 3, Offset 0
Zeile 22 bei "}": Entfernt wird b: Variable, Tiefe 2, Offset 0
Zeile 22 bei "}": Entfernt wird a: Variable, Tiefe 2, Offset 0
[Fehler 7] Zeile 23 bei "a": Parameter a doppelt
[Fehler 8] Zeile 24 bei "f": f ist kein Parameter
[Fehler 9] Zeile 26 bei "a": Parameter a zweimal deklariert
Zeile 27 bei "}": Entfernt wird f: Variable, Tiefe 2, Offset 0
Zeile 27 bei "}": Entfernt wird a: Variable, Tiefe 2, Offset 0
Zeile 27: Entfernt wird h: Funktion, Tiefe 1, Offset 0
Zeile 27: Entfernt wird f: Funktion, Tiefe 1, Offset 0
Zeile 27: Entfernt wird main: Funktion, Tiefe 1, Offset 0
Zeile 27: Entfernt wird g: undefinierte Funktion, Tiefe 1, Offset 0
[Fehler 10] Zeile 27: undefinierte Funktion g
```

5.4 Typdefinition für den Wert-Stack

Bei der Implementierung der Symboltabelle und der Semantikprüfungen haben wir häufig den Wert-Stack verwendet: für **Identifier**-Eingabesymbole haben wir Symboltabellenzeiger von der lexikalischen Analyse an den Parser übergeben; für **Constant**-Eingabesymbole haben wir Zeiger auf dynamisch erworbenen Speicher übergeben; die **parameter_list** haben wir mit Symboltabellenzeigern verkettet; die Anzahl von Ausdrücken in der **argument_list** haben wir auch auf dem Wert-Stack gezählt. Speziell diese letzte Verwendung von Stack-Elementen bedingt eine **union** von Datentypen für den Wert-Stack. Wir zählen mit einer **int**-Variable, aber wir zeigen auf die Symboltabelle mit einem Zeigertyp. Zwar können Zeiger in C in Zeiger auf beliebige Datentypen umgewandelt werden, aber es ist doch eine gute Idee, verschiedene Datentypangaben zu verwenden, wenn man auf Zeichenketten und Symboltabellenelemente zeigt.

Auch als **union** könnte der Wert-Stack immer noch so mit einem Typ versehen werden, wie das im Abschnitt 3.6 beschrieben wurde. Wenn wir aber dann auf Stack-Elemente in Aktionen mit der **$***i* Syntax zugreifen, müssen wir *yacc* informieren, welche Alternative der **union** wir in jedem Fall verwenden wollen. Anders formuliert: wir müssen einen Datentyp, der syntaktisch als Alternative der **union** ausgedrückt wird, mit all den Symbolen assoziieren, auf die wir über *yacc* mit **$***i* oder **$$** zugreifen. Dazu muß natürlich die bisher beschriebene *yacc*-Syntax etwas erweitert werden.

Wir ziehen es vor, unsere Grammatik in einem separaten Editierdurchgang mit Typen zu versehen, und zwar im Anschluß an die Konstruktion aller Aktionen. Bei diesem Durchgang notieren wir alle Eingabesymbole und Grammatikbegriffe, für die auf den Wert-Stack verwiesen wird, und wir legen einen Datentyp für das zugehörige Stack-Element fest. Verlassen wir uns darauf, daß die als Voreinstellung vorhandene Aktion

```
{ $$ = $1; }
```

wirklich einen Wert übergibt, müssen wir für die zugehörigen Symbole auch dann einen Datentyp festlegen, wenn diese Aktion nicht explizit angegeben ist. (Dies ist mit ein Grund, wieso wir normalerweise die Punkte in einer *yacc*-Eingabe kommentieren, wo wir uns auf diese Aktion verlassen.)

Haben wir die nötigen Datentypen bestimmt, zusammen mit den Symbolen für die Typen vereinbart werden müssen, können wir die Eingabe für *yacc* entsprechend modifizieren. Wir beschreiben den Vorgang mit der *kleinC*-Grammatik als konkretem Beispiel.

Zuerst legen wir den Datentyp der Stack-Elemente fest. Dies geschieht im ersten Teil der Eingabe für *yacc* im Stil einer **union**-Deklaration von C, der ein **%**-Zeichen vorausgeht. [1] In unserem Fall sind Wert-Stack-Elemente Zeiger auf die Symboltabelle, Zeiger auf Zeichenketten, und Integer-Werte als Zähler. Wir definieren:

```
%union  {
        struct symtab * y_sym;  /* Identifier */
        char * y_str;           /* Constant */
        int y_num;              /* Zaehler */
        }
```

Als nächstes vereinbaren wir einen Datentyp für die Eingabesymbole, für die bei der lexikalischen Analyse ein Wert an **yylval** zugewiesen wird. Syntaktisch wird dazu der Name einer **union**-Alternative, eingeschlossen in **< >**, zwischen **%token** und einer Liste von Eingabesymbolnamen eingefügt, die diesen Typ erhalten soll. Bei uns gehören zu **Identifier** und **Constant** entsprechende Werte auf dem Stack:

---

[1] Es gibt andere Möglichkeiten um diesen Datentyp festzulegen. Wir glauben aber, daß speziell diese Technik sowohl gut sichtbar ist in der Eingabe für *yacc*, als auch sehr bequem, denn die resultierende **union**-Deklaration sowie eine **extern**-Deklaration für **yylval** werden dann automatisch in die Datei *y.tab.h* eingefügt.

```
%token  <y_sym> Identifier
%token  <y_str> Constant
```

Die Werte werden an **yylval** in der Funktion **s_lookup()** zugewiesen. Auch bei diesen Zuweisungen muß natürlich jeweils die richtige **union**-Alternative verwendet werden (dies haben wir im Abschnitt 5.3 bereits stillschweigend so gemacht).

Schließlich müssen wir noch Datentypen für all die Grammatikbegriffe vereinbaren, für die ein Zugriff auf **$***i* oder **$$** erfolgt. Dazu wird eine **%type**-Definition verwendet, die sehr ähnlich ist zur **%token**-Definition für Eingabesymbole. Wir verwenden **argument_list** und **optional_argument_list** als Zähler und **parameter_list** und **optional_parameter_list** um den Anfang der Parameterkette weiterzureichen:

```
%type   <y_sym> optional_parameter_list, parameter_list
%type   <y_num> optional_argument_list, argument_list
```

Man beachte, daß *yacc* sehr streng prüft, daß *alle* Verweise auf den Wert-Stack korrekte Typen besitzen, wenn **%union**, **%type** oder die < > Syntax auch nur einmal verwendet wird. Jeder Verstoß sorgt dafür, daß *yacc* abgebrochen wird, mit einer Meldung, aus der die fehlerhafte Eingabezeile hervorgeht.

Es gibt noch eine subtile Möglichkeit Datentypen festzulegen. Diese wird speziell für anonyme Grammatikbegriffe (siehe Abschnitt 3.7) benötigt. Wir führen sie vor, wenn wir sie im Abschnitt 6.2 benötigen.

5.5 Aufgaben

1. Schreiben Sie ein Testprogramm in *kleinC*, das absichtlich Fehler enthält und alle Semantikfehlermeldungen im Compiler auslöst. ✏

2. Verwenden Sie ein *hash*-Verfahren für die *kleinC*-Symboltabelle, um die Suche nach Namen zu beschleunigen. Zeigen Sie, zum Beispiel mit Hilfe der *profile*-Möglichkeiten in C, daß das Resultat wirklich effizienter ist als unsere Lösung. ✏

3. Fügen Sie in *kleinC* einen Stack mit Block-Deskriptoren ein, der im wesentlichen auf dem *yacc*-Wert-Stack realisiert werden kann. Wie muß die Typvereinbarung für den Wert-Stack abgeändert werden? Versuchen Sie auch hier den Effizienzgewinn zu messen. ✏

4. Entfernen Sie die Teile der Symboltabellenverwaltung, die überflüssig sind, wenn ein Montierer verwendet wird. Entwerfen Sie, wie (und wann) die Information an den Montierer übergeben wird. ✏

5. Erweitern Sie den Tischrechner mit einfachen Operationen für Zeichenketten. Der Operator + soll Addition von numerischen Werten und Verkettung von Zeichenketten bedeuten. Soll das dadurch entstehende Problem in der Grammatik oder in separaten Semantikfunktionen behandelt werden? Das heißt: soll die Grammatik nur **Constant** kennen, oder lieber **Number** und **String**? ✏

Kapitel 6: Speicherverwaltung

Wir können jetzt eine *Implementierung* der *kleinC*-Sprache für eine bestimmte Zielmaschine definieren. Dazu legen wir die Grundsätze der *Speicherverwaltung* und der *Generierung von Befehlsfolgen* fest.

Interessanterweise haben wir die gesamte Spracherkennung realisiert ohne zu wissen, für welche Zielmaschine wir schließlich Code generieren würden. Diese Tatsache unterstreicht, daß ein großer Teil eines Compilers völlig maschinenunabhängig entwickelt werden kann.

In diesem und den folgenden Kapiteln werden wir uns auf die prinzipiellen Aspekte der Code-Generierung konzentrieren und nicht versuchen, mit den Details einer bestimmten Hardware-Architektur fertig zu werden. Wir beschreiben deshalb eine Implementierung für eine fiktive Zielmaschine, die unserer *kleinC*-Sprache besser angepaßt ist als existente Maschinen.

6.1 Grundbegriffe

Für die Speicherverwaltung definieren wir die Repräsentierung von Variablen, das heißt, die Implementierung von Vereinbarungen. Wir legen fest, wieviel Speicherplatz ein Objekt in Abhängigkeit von seinem Datentyp benötigt, und wie das Objekt adressiert werden soll, das heißt, wo sich das Objekt zur Laufzeit befinden soll, damit es eine bestimmte Lebensdauer erhält.

Die Laufzeit-Speicherverwaltung spiegelt bis zu einem gewissen Grad die Organisation der Symboltabelle zur Übersetzzeit wieder:

In Basic gibt es nur globale Variablen, die man in einer globalen Symboltabelle aufbewahrt, welche dann auch zur Laufzeit (bei einem Interpreter) die Werte der Variablen enthalten kann. Die Parameter einer vom Benutzer definierten Funktion (**def fn**) kann man auf dem gleichen Stack ablegen, der ohnedies zur Bewertung von Ausdrücken benötigt wird.

In Fortran kann man Unterprogramme nicht rekursiv aufrufen. Da die Werte von lokalen Variablen über alle Aufrufe einer Funktion hinweg erhalten bleiben müssen, müssen wir jeder Variablen in jedem Unterprogramm eigenen Speicherplatz zuordnen.

In C, Pascal und anderen Algol-artigen Sprachen kann man Unterprogramme rekursiv aufrufen. Für diese Sprachen müssen wir Speicherplatz für lokale Variablen dynamisch zuordnen, wenn ein Unterprogramm aufgerufen wird. Da die Werte lokaler Variablen normalerweise am Ende eines Unterprogrammaufrufs verlorengehen, können wir den reservierten Speicherplatz dann wieder weiterverwenden. Funktionen werden im Stil einer Stack-Disziplin aufgerufen, und diese Disziplin muß zum Management der lokalen Variablen ebenfalls eingehalten werden.

Die Geltungsbereiche von Namen sind in Algol-artigen Sprachen verschachtelt; dadurch ergibt sich ein weiteres Problem. Wenn Funktionsdefinitionen verschachtelt sein können, wie in Pascal, so hat eine Funktion jeweils nur Zugriff auf eine begrenzte Menge der lokalen Variablen der momentan aktiven Funktionen.

Sind Blöcke mit Definitionen verschachtelt, so ermöglichen die Regeln zur Geltungsdauer von lokalen Variablen in diesen Blöcken eine Wiederverwendung des Speicherplatzes, die bereits bei der Übersetzung arrangiert werden kann. Diese Blöcke, die sogenannten *compound statements*, kann man als anonyme Unterprogramme ansehen, die nur sequentiell aufgerufen werden, das heißt, genau in der Reihenfolge, in der sie der Compiler selbst sieht. Da diese Blöcke anonym sind, können sie auch nicht rekursiv aufgerufen werden; der Compiler hat also alle notwendigen Informationen zur Geltungsdauer von lokalen Variablen, die in einem solchen Block definiert sind.

Skalare Datentypen kann man in der Regel leicht repräsentieren – meistens korrespondieren sie zu Objekten, für die eine direkte Repräsentierung auf der Zielmaschine zusammen mit entsprechenden Maschinenbefehlen existiert. Eine seltene Ausnahme bilden die Aufzählungen, wie man sie etwa in Pascal oder C (**enum**) findet.

Die Repräsentierung von Datenstrukturen ist wesentlich schwieriger. Vektoren oder Matrizen als Sprachelemente bedingen, daß vom Compiler und zur Laufzeit arithmetische Operationen mit Adreßwerten durchgeführt werden. Bei einer Sprache wie APL benötigt man darüberhinaus auch noch recht aufwendige Mechanismen zur dynamischen Speicherverwaltung. Die Situation ist noch relativ einfach, wenn Vektoren feste, konstante Indexbereiche besitzen, wie in Pascal, oder wenigstens nicht dynamisch dimensioniert werden können, wie in Fortran oder C. Bei diesen Sprachen kann der Compiler Adressen auch für Vektoren festlegen – Vektoren werden dann einfach wie sehr große, skalare Objekte behandelt. Können Vektoren, wie in PL/I, dynamisch dimensioniert werden, kann erst zur Laufzeit Speicher bereitgestellt werden. Derartige Vektoren werden normalerweise mit Hilfe von (internen) Zeigern manipuliert.

Die Definition von Pascal schlägt vor, daß ein **record** mit einer **case**-Variante mit wechselndem Speicherbedarf implementiert werden sollte. Macht man dies nicht, so ist die Implementierung von Strukturen relativ leicht. Für die Komponenten muß man Adressen, relativ zum Anfang der Struktur, festlegen und dabei entsprechende Ausrichtungserfordernisse (*alignment*) beachten. Danach behandelt man eine Struktur als Ganzes wie ein großes skalares Objekt.

Zeiger, wenn auch 'ein Schritt zurück, von dem wir uns nie mehr erholen' (C. A. R. Hoare), repräsentiert man als Maschinenadressen. Probleme bei Zeigern ergeben sich aus der Tatsache, daß man aus semantischen Überlegungen heraus meist verlangt, daß ein Zeiger immer nur auf Objekte eines bestimmten Datentyps zeigt. Während man die Verwendung von Zeigern bei der Übersetzung durchaus überprüfen kann, ist es selbst zur Laufzeit fast unmöglich, die Gültigkeit von Zeigerwerten

(Adressen) entsprechend zu überwachen. Enthält eine Sprache Zeiger-Datentypen, so existiert fast immer auch eine dynamische Speicherverwaltung. Eine Funktion wie **calloc()** oder **new()**, die neue Speicherflächen bereitstellt, realisiert man sehr leicht - in erster Näherung teilt man einfach sukzessiv Abschnitte aus einem großen Vektor zu. Wenn aber erst eine Funktion wie **free()** oder **dispose()** zur Verfügung steht, mit der Speicher explizit zur Wiederverwendung markiert wird, kann die Gültigkeit von Zeigerwerten nicht mehr ohne weiteres verifiziert werden.

Betrachten wir nun die wesentlichen Aspekte der Speicherverwaltung für eine Sprache im Stil von C etwas näher: wir müssen die Repräsentierung von Datentypen festlegen sowie die Techniken zur Speicherverwaltung innerhalb von statisch verschachtelten Blöcken und bei dynamisch verschachtelten Funktionsaufrufen.

Bei *kleinC* ist die erste Entscheidung leicht, da wir nur Integer-Variablen zulassen: jedes Objekt benötigt ein Wort im Hauptspeicher.

Die Adressierung von Objekten ist etwas diffiziler, da globale und lokale Objekte existieren und da Funktionen rekursiv aufgerufen werden können. Darüberhinaus herrschen sehr lockere Sitten bei der Übergabe von Parametern: die Anzahl der Formalparameter muß nicht mit der Anzahl der tatsächlich übergebenen Argumente übereinstimmen; das Programm muß jedoch korrekt funktionieren, solange nur auf Parameter zugegriffen wird, für die auch Argumente übergeben wurden.

Als allgemeine Lösung kombiniert man die lokalen Variablen jeder Funktion in einem *Prozedursegment* (*activation record*) und unterhält zur Laufzeit einen Stack derartiger Segmente parallel zu den Funktionsaufrufen. Während der Übersetzung kann man ein globales Segment für die globalen Variablen konstruieren, das dann gleichzeitig das erste (und permanente) Element auf dem Stack zur Laufzeit ist.

Generieren wir Code für eine Stack-Maschine, so können wir den Stack der Maschine selbst für die Prozedursegmente verwenden sowie zur Bewertung von arithmetischen Ausdrücken und zur Übergabe von Argumenten an Unterprogramme. Die Argumente werden auf diesem Stack abgelegt, *bevor* das Prozedursegment für die aufgerufene Funktion angelegt wird, und zwar *in umgekehrter Reihenfolge*, damit ihre Laufzeit-Position relativ zum neuen Prozedursegment bei der Code-Generierung bekannt ist, auch wenn ihre Zahl selbst noch nicht fixiert ist.

Die Umkehrung der Argumentreihenfolge erweist sich als Komplikation - die Argumentausdrücke müssen bei der Übersetzung temporär gespeichert werden. 'Zum Glück' verfügt unsere fiktive Maschine im wesentlichen über ein separates Segment zur Übergabe von Parametern.

Variablen werden im Prozedursegment repräsentiert, indem man eindeutige Adressen relativ zum Anfang des Segments festlegt und diese in der Symboltabelle ablegt, zwecks späterer Verwendung bei der Code-Generierung. Man muß dann noch dafür sorgen, daß die Anfangsadressen der aktiven Prozedursegmente zur Laufzeit entsprechend in Registern verfügbar sind.

Bei Sprachen wie C benötigen wir nur drei solche Anfangsadressen, nämlich für das globale Segment, für den Parameterbereich und für das aktuelle Prozedursegment. Bei Algol-artigen Sprachen, bei denen Funktionsdefinitionen verschachtelt sein können, ist Zugriff auf viele Prozedursegmente gleichzeitig möglich, oft nötig und nicht gerade leicht zu realisieren. Die Dinge werden noch wesentlich komplizierter, wenn in einer Algol-artigen Sprache Funktionen als Parameter übergeben werden können: die Adressen der Prozedursegmente, auf die die übergebene Funktion zugreifen kann, der sogenannte *display* der Funktion, sind implizit ein Teil der als Parameter übergebenen Funktion. Dies erklärt, warum Funktionsaufrufe in Pascal wesentlich aufwendigere Operationen sind als in C.

Wird eine Funktion aufgerufen, muß der alte *display* in Stack-Manier gespeichert werden, damit er bei Rückkehr vom Funktionsaufruf wiederhergestellt werden kann. Der *display* wird dadurch Teil der Rücksprungadresse der Funktion. Für C oder *kleinC* müssen wir nur die Adresse des vorhergehenden Prozedursegments speichern, da auf das globale Segment immer zugegriffen werden kann.

Wie viele andere Sprachen erlaubt auch *kleinC* Definitionen in verschachtelten Blöcken, die dann eingeschränkten Geltungsbereich und eingeschränkte Lebensdauer besitzen. Dadurch werden keine zusätzlichen Segmente zur Laufzeit benötigt. Während der Übersetzung verwenden wir einen Zähler, den wir über verschachtelte Blöcke hinweg unterhalten, und von dem wir die lokalen, relativen Adressen im Prozedursegment ableiten.

Dieser lokale Zähler wird mit Null initialisiert, wenn wir einen Funktionskopf bearbeiten. Zusätzlich bestimmen wir auch den maximalen Wert, den dieser Zähler innerhalb eines Prozedursegments erreicht. Am Anfang eines Blocks legen wir den aktuellen Wert des Zählers auf einem Stack ab. Anschließend wird der Zähler ganz normal für jede Definition einer Variablen weiter inkrementiert. Wird das Ende eines Blocks erreicht, vergleicht man den Zähler mit dem bisher bekannten Maximum, und falls nötig wird ein neues Maximum notiert. Anschließend setzt man den Zähler auf den Wert, der am Anfang des Blocks auf dem Stack abgelegt wurde. Auf diese Weise belegen Variablen, die auf der *gleichen* Verschachtelungstiefe in zwei Blöcken definiert wurden, die gleichen Speicherbereiche (zu verschiedenen Zeitpunkten).

Wenn wir ein Prozedursegment abschließen, kennen wir jeweils seine Größe und können später Code generieren um zur Laufzeit dynamisch genügend Speicher zur Verfügung zu stellen.

In C kann man globale Datendefinitionen und Funktionsdefinitionen in beliebiger Reihenfolge angeben. Wir unterhalten deshalb einen separaten, globalen Zähler. Dieser Zähler wird geeignet initialisiert und bei jeder globalen Definition entsprechend inkrementiert.

Parameternamen werden bei Bearbeitung einer Parameterliste verkettet. Deklarationen für die Parameter werden anschließend bearbeitet - verschiedene Parameter brauchen möglicherweise verschieden viel Speicherplatz, je nach Deklaration. Sind alle Parameterdeklarationen einer Funktion abgearbeitet worden, kann man die

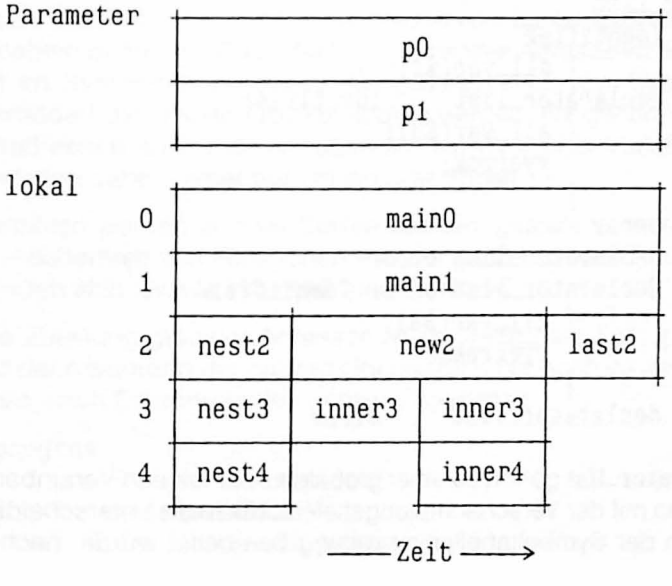
Parameterkette verfolgen und die nötigen, relativen Adressen definieren; die Reihenfolge der Parameterkette, also der Parameterliste, bestimmt dabei die Reihenfolge der relativen Adressen und die Deklarationen beeinflussen den jeweiligen Platzbedarf.

6.2 Beispiel

Das folgende Beispiel demonstriert, wie Teile des Prozedursegments von Definitionen in parallelen Blöcken gemeinsam genutzt werden:

```
/*
 *      Speicherverwaltung -- insbesonders verschachtelte Bloecke
 */

main(p0, p1)
{       int main0, main1;
        {       int nest2, nest3, nest4;
        }
        {       int new2;
                {       int inner3;
                }
                {       int inner3, inner4;
                }
        }
        {       int last2;
        }
}
```

Für diese Funktion sollte folgendes Prozedursegment konstruiert werden:



Wenn unser Compiler mit Speicherverwaltung dieses Beispiel bearbeitet, entsteht folgende Ausgabe:

```
Zeile 6 bei "{": Parameterbereich umfasst 2 Wort(e)
Zeile 8 bei "}": Entfernt wird nest4: Variable, Tiefe 4, Offset 4
Zeile 8 bei "}": Entfernt wird nest3: Variable, Tiefe 4, Offset 3
Zeile 8 bei "}": Entfernt wird nest2: Variable, Tiefe 4, Offset 2
Zeile 11 bei "}": Entfernt wird inner3: Variable, Tiefe 5, Offset 3
Zeile 13 bei "}": Entfernt wird inner4: Variable, Tiefe 5, Offset 4
Zeile 13 bei "}": Entfernt wird inner3: Variable, Tiefe 5, Offset 3
Zeile 14 bei "}": Entfernt wird new2: Variable, Tiefe 4, Offset 2
Zeile 16 bei "}": Entfernt wird last2: Variable, Tiefe 4, Offset 2
Zeile 17 bei "}": Entfernt wird main1: Variable, Tiefe 3, Offset 1
Zeile 17 bei "}": Entfernt wird main0: Variable, Tiefe 3, Offset 0
Zeile 17 bei "}": Entfernt wird p1: Variable, Tiefe 2, Offset 1
Zeile 17 bei "}": Entfernt wird p0: Variable, Tiefe 2, Offset 0
Zeile 17 bei "}": lokaler Bereich umfasst 5 Wort(e)
Zeile 17: Entfernt wird main: Funktion, Tiefe 1, Offset 0
Zeile 17: globaler Bereich umfasst 1 Wort(e)
```

Diese Ausgabe stammt von der Ablaufverfolgung, die als Teil der Symboltabellenverwaltung installiert wurde: die relativen Adressen werden in der **s_offset**-Komponente eines Symboltabelleneintrags abgelegt, und diese Komponente wird im Rahmen der Ablaufverfolgung ausgegeben.

Betrachten wir die Aktionen, die sich mit der Speicherverwaltung beschäftigen. Wenn ein Variablenname in einer **declarator_list** auftaucht, muß eine Adresse im globalen oder lokalen Bereich zugeteilt werden:

```
declarator_list
        : Identifier
                { all_var($1); }
        | declarator_list ',' Identifier
                { all_var($3);
                  yyerrok;
                }
        | error
        | declarator_list error
        | declarator_list error Identifier
                { all_var($3);
                  yyerrok;
                }
        | declarator_list ',' error
```

Diese **declarator_list** gehört zu einer globalen oder lokalen Vereinbarung. Die zwei Fälle kann man mit der Verschachtelungstiefe **s_blknum** unterscheiden, wenn der **Identifier** von der Symboltabellenverwaltung bearbeitet wurde: nachdem der

Eintrag von der Funktion **make_var()** initialisiert wurde, sollten wir weder die Verschachtelungstiefe **2** vorfinden, also den Wert für einen Parameternamen, noch die Tiefe **0**, also den Wert für einen undefinierten Symboltabelleneintrag.

```
int     g_offset = 1,    /* Offset im globalen Bereich */
        l_offset = 0,    /* Offset im lokalen Bereich */
        l_max;           /* Groesse des lokalen Bereichs */

all_var(symbol)
        register struct symtab * symbol;
{       extern struct symtab * make_var();

        symbol = make_var(symbol);

        /* falls kein Parameter:
           im entsprechenden Bereich anlegen */
        switch (symbol->s_blknum) {
        default:                          /* lokal */
                symbol->s_offset = l_offset++;
        case 2:                           /* Parameter */
                break;
        case 1:                           /* global */
                symbol->s_offset = g_offset++;
                break;
        case 0:
                bug("all_var");
        }
}
```

Für lokale Variablen wird die relative Adresse vom Zähler **l_offset** festgelegt, für globale Variablen definiert sie **g_offset**. Ist die relative Adresse im richtigen Segment in **s_offset** im Symboltabelleneintrag festgehalten, muß zum betroffenen Zähler der Speicherbedarf des neuen Objekts addiert werden. Wir gehen von einer Maschine mit Wortadressen aus, und wir haben es nur mit Integer-Variablen zu tun; unsere Zähler werden daher immer nur um eins vergrößert.

Variablen werden an zwei Stellen definiert: globale Variablendefinitionen können wir - außerhalb von Funktionen - in der ganzen Quelle finden, lokale Definitionen befinden sich jeweils innerhalb einer Funktion.

Die Zuteilung globaler Adressen ist der einfachere Fall: **g_offset** wird initialisiert und dann während der ganzen Übersetzung nur noch inkrementiert. Am Schluß der Quelle, nach Erkennung des ganzen Programms

```
program
        :       { init(); }
          definitions
                { all_program(); }
```

finden wir die Größe des globalen Bereichs in **g_offset**:

```
all_program()
{
        blk_pop();

#ifdef  TRACE
        message("globaler Bereich umfasst %d Wort(e)",
                g_offset);
#endif
}
```

Wir werden das erste Wort im globalen Bereich zur Übergabe von Funktionswerten verwenden; folglich initialisieren wir **g_offset** mit **1** und nicht mit **0**.

Für Prozedursegmente müssen wir **l_offset** in einer Stack-Disziplin verwalten, während wir verschachtelte Blöcke bearbeiten:

```
compound_statement
        : '{'
                { $<y_lab>$ = l_offset;
                  blk_push();
                }
          declarations statements rr
                { if (l_offset > l_max)
                        l_max = l_offset;
                  l_offset = $<y_lab>2;
                  blk_pop();
                }
```

Hier sieht man, daß eine eingebettete Aktion ebenfalls einen Wert auf dem *yacc*-Wert-Stack ablegen kann, nämlich auch durch Zuweisung an **$$**; der Wert kann dann später genauso nach Position als **$2** verwendet werden, wie wenn diese Aktion ein Symbol gewesen wäre. Haben wir einen Typ für den Wert-Stack vereinbart, so müssen wir Verweise auf das zu der eingebetteten Aktion gehörende Stack-Element explizit mit einem Datentyp versehen. Das Beispiel zeigt, daß dies dadurch geschieht, daß man die Typinformation direkt im Verweis angibt.

In diesem Fall benötigen wir einen neuen Datentyp für den Wert-Stack, nämlich eine Marke. Wir müssen deshalb die Variante, die den Wert-Stack beschreibt, noch um eine Alternative erweitern:

```
%union  {
        struct symtab * y_sym;  /* Identifier */
        char * y_str;           /* Constant */
        int y_num;              /* Zaehler */
        int y_lab;              /* Marke */
        }
```

Beginnt die Verarbeitung von **compound_statement**, so definiert **l_offset** den Punkt im Prozedursegment, der als relative Adresse der ersten lokalen Variablen in diesem **compound_statement** verwendet werden soll. Beim äußersten **compound_statement** jeder Funktion ist dies der Anfangswert von **l_offset**, das heißt **0**.

Diese erste relative Adresse wird auf dem *yacc*-Wert-Stack aufbewahrt. Wird das **compound_statement** abgeschlossen, so kann der alte Wert von **l_offset** entsprechend wiederhergestellt werden, und damit wird die für das **compound_statement** im Prozedursegment verwendete Fläche erneut verfügbar.

Da wir die Größe des Prozedursegments einer Funktion feststellen müssen, müssen wir den Maximalwert von **l_offset** bestimmen: die Größe **l_max** des Prozedursegments ist der Maximalwert, den **l_offset** über alle in die Funktion verschachtelten Blöcke hinweg je erreicht. Bevor wir also das äußerste **compound_statement** einer Funktion bearbeiten, initialisieren wir **l_max** auf **0**:

```
function_definition
        : Identifier '('
                { make_func($1);
                  blk_push();
                }
          optional_parameter_list rp
          parameter_declarations
                { chk_parm($1, parm_default($4));
                  all_parm($4);
                  l_max = 0;
                }
          compound_statement
                { all_func($1); }
```

Am Schluß der Funktion enthält **l_max** den gewünschten Wert:

```
all_func(symbol)
        struct symtab * symbol;
{
        blk_pop();

#ifdef  TRACE
        message("lokaler Bereich umfasst %d Wort(e)", l_max);
#endif
}
```

Die relative Adresse eines Parameters hängt von der Position des Parameters in der **parameter_list** ab sowie von dem Datentyp des Parameters, falls eine Deklaration angegeben wurde. Wir können deshalb diese relativen Adressen nicht im Zuge der Bearbeitung der **parameter_list** festlegen. Erst wenn alle **parameter_declarations** abgehandelt und die nötige **int**-Vereinbarung für die

restlichen Parameter getroffen wurde, können wir der Kette durch die Symboltabelle folgen, die alle Parameter einer Funktion verbindet, und alle relativen Adressen zusammen definieren:

```
all_parm(symbol)
        register struct symtab * symbol;
{       register int p_offset = 0;

        while (symbol)
        {       symbol->s_offset = p_offset ++;
                symbol = symbol->s_plist;
        }

#ifdef  TRACE
        message("Parameterbereich umfasst %d Wort(e)",
                p_offset);
#endif
}
```

Wo wir die Argumentwerte zur Laufzeit antreffen wollen, hängt ab von der Reihenfolge, in der die Symboltabelleneinträge verkettet wurden, und davon, ob wir **p_offset** von Null ab durch positive Adressen in einem separaten Parametersegment inkrementieren, oder von Null ab durch negative Adressen relativ zum aktuellen Prozedursegment dekrementieren.

In dieser Implementierung werden wir die Argumentwerte für eine Funktion der Reihe nach von links nach rechts auf den Stack bringen. Die Adresse des ersten Arguments wird als Anfangsadresse eines separaten Parametersegments für den Funktionsaufruf notiert. Die Parameter erhalten Adressen relativ zu dieser Anfangsadresse, und müssen folglich von Null ab in der Reihenfolge der **parameter_list** von links nach rechts ansteigen. Wir verketten also die Parameter von links nach rechts; wenn wir dann die Parameterkette verfolgen, inkrementieren wir **p_offset** von Null ab.

Leider haben wir bisher **parameter_list** links-rekursiv definiert:

```
parameter_list
        : Identifier
                { $$ = link_parm($1, (struct symtab *) 0); }
        | parameter_list ',' Identifier
                { $$ = link_parm($3, $1);
                  yyerrok;
                }
        | error
                { $$ = 0; }
```

```
        | parameter_list error
        | parameter_list error Identifier
                { $$ = link_parm($3, $1);
                  yyerrok;
                }
        | parameter_list ',' error
```

Dies bedeutet aber, daß die Liste von links nach rechts *zusammengefaßt* wird, das heißt, nach dieser Regel verläuft die Parameterkette dann gerade von rechts nach links!

Die Lösung sollte selbstverständlich sein: die Verkettung erfolgt, wenn wir **Identifier** als **parameter_list** reduzieren. Wir müssen sicherstellen, daß dies von rechts nach links erfolgt, das heißt, daß wir einen neuen **Identifier** *vor* einer schon verketteten **parameter_list** einfügen. Dies geschieht ganz genau dann, wenn wir rechts-rekursiv formulieren!

```
parameter_list
        : Identifier
                { $$ = link_parm($1, (struct symtab *) 0); }
        | Identifier ',' parameter_list
                { $$ = link_parm($1, $3);
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | error parameter_list
                { $$ = $2; }
        | Identifier error parameter_list
                { $$ = link_parm($1, $3); }
        | error ',' parameter_list
                { $$ = $3;
                  yyerrok;
                }
```

Die Grammatik kann hier sehr leicht modifiziert werden, ohne bösartige Nebeneffekte.[1] Warum kann wohl dieser Trick *nicht* dazu dienen, die Reihenfolge umzukehren, in der die Argumentwerte berechnet werden?

Das Beispiel aus dem vorigen Kapitel illustriert jetzt, was passiert, wenn die Quelle Fehler enthält:

---

[1] Abgesehen vom Platzbedarf auf dem *yacc*-Stack! A.d.Ü.

```
/*
 *      Symboltabellen-Demonstration, mit Fehlerbehandlung
 */

main(a,b) int a,b;
{       a=b;
        {a;b;}
        if (a==b) {a;b;}
        if (a==b+1) a; else b;
        while (a==b) {a; break;}
        return;
}
int f() { int x; int y; return x+y; }
f(a,b)                  /* 14: Funktion zweimal definiert */
        int a;          /* undeklarierter Parameter b (kein Fehler) */
{       int b,c;        /* 16: Name b dupliziert Parameter b */
        int c,d;        /* 17: Name c zweimal vereinbart */
        e=b;            /* 18: undefinierte Variable e */
        f=c+d;          /* 19: Funktion als Variable verwendet */
        a(b);           /* 20: Variable als Funktion verwendet */
        g();            /* 21: undefinierte Funktion g */
}
h(a,a)                  /* 23: Parameter doppelt */
        int f;          /* 24: kein Parameter */
        int a;
        int a;          /* 26: Parameter zweimal */
{ }
```

Mit Speicherverwaltung entsteht folgende Ausgabe:

```
Zeile 6 bei "{": Parameterbereich umfasst 2 Wort(e)
Zeile 12 bei "}": Entfernt wird b: Variable, Tiefe 2, Offset 1
Zeile 12 bei "}": Entfernt wird a: Variable, Tiefe 2, Offset 0
Zeile 12 bei "}": lokaler Bereich umfasst 0 Wort(e)
Zeile 13 bei "{": Parameterbereich umfasst 0 Wort(e)
Zeile 13 bei "}": Entfernt wird y: Variable, Tiefe 3, Offset 1
Zeile 13 bei "}": Entfernt wird x: Variable, Tiefe 3, Offset 0
Zeile 13 bei "}": lokaler Bereich umfasst 2 Wort(e)
[Fehler 1] Zeile 14 bei "(": Function f zweimal definiert
[Warnung] Zeile 16 bei "{": Funktion f hat eigentlich 0 Argument(e)
Zeile 16 bei "{": Parameterbereich umfasst 2 Wort(e)
[Fehler 2] Zeile 16 bei "b": Name b zweimal vereinbart
[Fehler 3] Zeile 17 bei "c": Name c zweimal vereinbart
[Fehler 4] Zeile 18 bei ";": undefinierte Variable e
[Fehler 5] Zeile 19 bei ";": Funktion f als Variable verwendet
```

```
[Fehler 6] Zeile 20 bei "(": Variable a als Funktion verwendet
Zeile 22 bei "}": Entfernt wird e: Variable, Tiefe 3, Offset 0
Zeile 22 bei "}": Entfernt wird d: Variable, Tiefe 3, Offset 3
Zeile 22 bei "}": Entfernt wird c: Variable, Tiefe 3, Offset 2
Zeile 22 bei "}": Entfernt wird c: Variable, Tiefe 3, Offset 1
Zeile 22 bei "}": Entfernt wird b: Variable, Tiefe 3, Offset 0
Zeile 22 bei "}": Entfernt wird b: Variable, Tiefe 2, Offset 1
Zeile 22 bei "}": Entfernt wird a: Variable, Tiefe 2, Offset 0
Zeile 22 bei "}": lokaler Bereich umfasst 4 Wort(e)
[Fehler 7] Zeile 23 bei ")": Parameter a doppelt
[Fehler 8] Zeile 24 bei "f": f ist kein Parameter
[Fehler 9] Zeile 26 bei "a": Parameter a zweimal deklariert
Zeile 27 bei "{": Parameterbereich umfasst 1 Wort(e)
Zeile 27 bei "}": Entfernt wird f: Variable, Tiefe 2, Offset 0
Zeile 27 bei "}": Entfernt wird a: Variable, Tiefe 2, Offset 0
Zeile 27 bei "}": lokaler Bereich umfasst 0 Wort(e)
Zeile 27: Entfernt wird h: Funktion, Tiefe 1, Offset 0
Zeile 27: Entfernt wird f: Funktion, Tiefe 1, Offset 0
Zeile 27: Entfernt wird main: Funktion, Tiefe 1, Offset 0
Zeile 27: Entfernt wird g: undefinierte Funktion, Tiefe 1, Offset 0
[Fehler 10] Zeile 27: undefinierte Funktion g
Zeile 27: globaler Bereich umfasst 1 Wort(e)
```

Man sieht, daß für die nicht vereinbarten Variablen auch keine Adressen zugeteilt werden, und daß doppelt vereinbarte Variablennamen für ein gewisses Durcheinander bei den relativen Adressen sorgen. Wenn ein Programm Fehler enthält, ist der resultierende Code natürlich nicht mehr unbedingt korrekt, aber wir müssen trotzdem dafür sorgen, daß der Compiler nicht unvorhergesehen oder unfreundlich terminiert.

6.3 Aufgaben

1. Nehmen wir an, daß unser Compiler seine Ausgabe einem Assembler liefert, das heißt, unser Compiler produziert Assembler-Text und nicht nur Maschinencode; wie könnte man dann globale Variable mit ihren eigenen Namen vereinbaren? Würde dies den Compiler vereinfachen? ✏

2. Nehmen wir an, daß *kleinC* für eine Maschine implementiert wird, bei der ein Wort mehr als nur einen Speicherplatz benötigt, das heißt, daß ein Wort etwa aus zwei separat adressierbaren Bytes besteht; wie müssen dann die Funktionen der Speicherverwaltung abgeändert werden? ✏

3. Viele Sprachen erlauben das Verschachteln von Funktionsdefinitionen. Welcher Teil der *kleinC*-Speicherverwaltung muß geändert werden, wenn Funktionsdefinitionen verschachtelt werden sollen? Beachten Sie dabei, daß eine Variable in einem weiter außen liegenden Block einer verschachtelten Funktion als globale Variable

zur Verfügung steht (solange ihr Name nicht in der inneren Funktion neu vereinbart wird). Wie kann die innere Funktion dann auch auf solche globalen Variablen zugreifen? ✏

4. Mit Hilfe einer rechts-rekursiven Formulierung konnten wir die Reihenfolge, in der die Namen in einer **parameter_list** erkannt und verkettet wurden, sehr leicht umkehren. Was passiert, wenn wir **argument_list** rechts-rekursiv formulieren? Eine rechts-rekursive Formulierung führt hier nicht zum Ziel. Welche andere Technik könnte man verwenden, damit die Argumentwerte für einen Funktionsaufruf in umgekehrter Reihenfolge, also von rechts nach links, auf dem Stack abgelegt werden? ✏

Kapitel 7: Code-Generierung

Nachdem jetzt die Speicherverwaltung für unsere fiktive Maschine geklärt ist, können wir einen Compiler produzieren, der auch den zugehörigen Code generiert. Der größte Teil dieses Kapitels befaßt sich mit der Code-Generierung für eine Stack-Maschine. Abschnitt 7.2 beginnt mit der Definition der Maschinenbefehle unserer *kleinC*-Maschine. Anschließend sehen wir, wie man Code für arithmetische Ausdrücke und für Zuweisungen erzeugt. Zum Schluß untersuchen wir die Probleme bei der Implementierung der **if**- und **while**-Kontrollstrukturen. Um unser Vorgehen einigermaßen überschaubar zu halten, versuchen wir kaum, den generierten Code zu optimieren. Einige Aufgaben am Schluß des Kapitels enthalten jedoch Hinweise, wie man etwas zusätzliche Effizienz erreichen könnte.

7.1 Grundbegriffe

Zum Entwurf der Code-Generierung legt man grundsätzlich fest, welche Maschinenbefehle zur Realisierung jeder Aktion verwendet werden, die man in der Programmiersprache ausdrücken kann. Als direkte Methode, wenigstens bei einer Stack-Maschine, bietet sich an, arithmetische Ausdrücke einfach in Postfix [1] umzuwandeln. Je nach Adreßarchitektur der Zielmaschine ist es unter Umständen schwierig, die lokalen und globalen Variablen adressierbar zu machen.

Eigentlich der einzige schwierige Aspekt, der sehr wesentlich ist für das Gelingen einer realistischen Implementierung, ist der Entwurf der Befehlsfolge zum Aufruf von Funktionen. In unserem Beispiel nehmen wir an, daß 'geeignete' Maschinenbefehle existieren; bei vielen Maschinen benötigt man sonst zum Management der Prozedursegmente entsprechende Unterprogramme, die jeweils mit einem übersetzten Programm montiert werden. Derartige Unterprogramme bieten sich übrigens dazu an, eine Ablaufverfolgung für Fehlersuche oder Statistik im übersetzten Programm zu installieren.

Zur Implementierung von Kontrollstrukturen wie Schleifen und Entscheidungen benötigt man normalerweise eine Reihe von Vorwärtssprüngen, zum Beispiel vorbei am **else**-Zweig einer **if**-Anweisung. Leider muß man diese Vorwärtssprünge meistens zweimal ausgeben: zuerst mit unbestimmter Zieladresse, damit Platz für den Sprungbefehl reserviert wird, und später mit der korrekten Zieladresse. Wenn wir das übersetzte Programm direkt im Speicher konstruieren, müssen wir die entsprechenden Speicherzellen tatsächlich korrigieren.

Wenn wir Assembler-Text erzeugen, könnten wir mit einer **origin**-Assembler-Anweisung den Assembler später auf die richtige Programmadresse einstellen, damit wir den Sprungbefehl nochmals mit der tatsächlichen Zieladresse ausgeben können. Wesentlich einfacher implementiert man Vorwärtssprünge natürlich, indem man eine Marke generiert und den Assembler den Vorwärtsverweis auflösen läßt.

---

[1] Umgekehrte polnische Notation, das heißt, die Operatoren folgen ihren Operanden.

Die **break**- und **continue**-Anweisungen in *kleinC* realisiert man jeweils mit einem Stack, auf dem bei der Bearbeitung von **while** oder ähnlichen Anweisungen geeignete Marken abgelegt werden. Spätestens wenn **switch** zu *kleinC* hinzugefügt wird, braucht man zwei Stacks, denn dann kann die für **continue** bestimmte Information sehr stark von der für **break** abweichen. Mit Hilfe der Stacks entscheiden wir jetzt auch, ob **break** und **continue** korrekt verwendet werden; diesen Teil der Semantikprüfung haben wir im Kapitel 5 bewußt aufgeschoben.

7.2 Beispiel

Wir beschreiben, wie man Assembler-Text für die *kleinC*-Maschine generiert. Das Assembler-Quellformat soll dem vieler existenter Assembler gleichen: je Zeile steht eine Anweisung oder ein Maschinenbefehl; in der ersten Spalte beginnt das optionale Feld für eine Marke; im zweiten Feld muß ein Befehlsname stehen; im dritten Feld kann eine Bereichsangabe oder andere Information angegeben werden; die Felder werden jeweils durch Zwischenraumzeichen getrennt; am Schluß der Zeile kann ein Kommentar folgen. Das Quellformat ist momentan recht unwichtig, da die Funktionen der Code-Generierung in dieser Hinsicht sehr leicht geändert werden können.

Als grobe Übersicht der Maschinenbefehle dient uns die Definition der Befehlsnamen in der Definitionsdatei *gen.h*:

```
/*
 *      Befehlsnamen fuer fiktive Maschine
 */

                    /* Befehl */   /* Operanden              */
#define OP_ALU      "alu"          /* Arithmetik-Logik-Op    */
#define OP_DEC      "dec"          /* Bereich,Offset         */
#define OP_INC      "inc"          /* Bereich,Offset         */
#define OP_LOAD     "load"         /* Bereich,Offset         */
#define OP_STORE    "store"        /* Bereich,Offset         */
#define OP_POP      "pop"          /*                        */
#define OP_JUMPZ    "jumpz"        /* Marke                  */
#define OP_JUMP     "jump"         /* Marke                  */
#define OP_CALL     "call"         /* Anz-Argum,Adresse      */
#define OP_ENTRY    "entry"        /* lokaler-Bereich        */
#define OP_RETURN   "return"       /*                        */

/*
 *      Bereichsnamen
 */

#define MOD_GLOBAL  "gbl"          /* globale Variable       */
#define MOD_PARAM   "par"          /* Parameter              */
```

```
#define MOD_LOCAL  "lcl"        /* lokale Variable      */
#define MOD_IMMED  "con"        /* "load": Constant     */

/*
 *      OP_ALU Namen
 */

#define ALU_ADD "+"             /* Addition             */
#define ALU_SUB "-"             /* Subtraktion          */
#define ALU_MUL "*"             /* Multiplikation       */
#define ALU_DIV "/"             /* Division             */
#define ALU_MOD "%"             /* Rest                 */
#define ALU_LT  "<"             /* Vergleich:   <       */
#define ALU_GT  ">"             /*              >       */
#define ALU_LE  "<="            /*              <=      */
#define ALU_GE  ">="            /*              >=      */
#define ALU_EQ  "=="            /*              ==      */
#define ALU_NE  "!="            /*              !=      */
#define ALU_AND "&"             /* Bit-UND              */
#define ALU_OR  "|"             /* Bit-ODER             */
#define ALU_XOR "^"             /* Bit-ODER exklusiv    */

/*
 *      Typvereinbarungen fuer Funktionen
 */

char * gen_mod();               /* Bereichsname */
```

Die einzelnen Maschinenbefehle definieren wir exakt, wenn wir im Kapitel 8 einen Simulator implementieren. Bis dahin genügt uns für die Code-Generierung eine kurze Skizze der Eigenschaften der *kleinC*-Maschine:

Die *kleinC*-Maschine ist eine Stack-Maschine. Für arithmetische Ausdrücke können wir daher sehr leicht Code generieren: der Wert einer Variablen oder Konstanten muß mit Hilfe des speziell dafür vorhandenen **load**-Maschinenbefehls auf den Stack gebracht werden, und für einen Operator muß ein entsprechender **alu**-Befehl generiert werden. Zum **alu**-Maschinenbefehl gehört ein Parameter, aus dem hervorgeht, wie die zwei obersten Werte auf dem Stack arithmetisch oder logisch verknüpft werden sollen. Das Resultat der Operation ersetzt dann die zwei Werte auf dem Stack. Erfreulicherweise existiert für jeden *kleinC*-Operator ein entsprechender **alu**-Parameter!

Zuweisungen werden natürlich mit **store**-Befehlen realisiert. In *kleinC* kann jedoch die Zuweisung als Operator verwendet werden, das heißt, der zugewiesene Wert kann noch weiter in einem Ausdruck verwendet werden. Der **store**-Maschinenbefehl entfernt deshalb den zugewiesenen Wert *nicht* vom Stack.

Dies geschieht statt dessen mit Hilfe eines **pop**-Befehls, der immer generiert werden muß, wenn der Wert eines Ausdrucks vom Stack entfernt werden soll.

Für **if**- und **while**-Anweisungen müssen geeignete Sprungbefehle generiert werden. Dabei entstehen auch Vorwärtssprünge, zum Beispiel zum **else**-Zweig einer **if**-Anweisung. Dazu werden eindeutige Marken mit Hilfe eines Zählers konstruiert. Die Marken werden dann auf dem Wert-Stack von *yacc* weitergegeben und schließlich vom Assembler, nach der Übersetzung, aufgelöst.

break und **continue** bieten ein schwierigeres Problem: in *kleinC* dürfen sie nur innerhalb von einer **while**-Schleife auftreten. Wir überwachen dies am besten mit Hilfe von zwei separaten Stacks, auf denen jede **while**-Anweisung entsprechende Marken deponiert, die am Schluß der abhängigen Anweisung wieder entfernt und geeignet definiert werden.

Einen Funktionsaufruf realisieren wir mit einem **call**-Maschinenbefehl. Die Funktionsargumente müssen sich dabei schon auf dem Stack befinden. Der **call**-Befehl hat als Parameter die Anzahl der bei diesem Funktionsaufruf tatsächlich übergebenen Argumente, damit der Parameterbereich für die aufgerufene Funktion entsprechend definiert werden kann.

Am Anfang einer Funktion müssen wir einen **entry**-Maschinenbefehl codieren, der angibt, wieviel Platz auf dem Stack für die lokalen Variablen reserviert werden muß. Wir gehen davon aus, daß **call** und **entry** zusammen für korrekte Parameterübergabe und dynamische Bereitstellung der lokalen Variablen sorgen.

Der **return**-Maschinenbefehl entfernt das Prozedursegment mit den lokalen Variablen wieder vom Stack und korrigiert alle relevanten Register. Falls bei **return** ein Resultatwert übergeben wird, das heißt, falls vor **return** noch ein Ausdruck bewertet wird, muß der Wert vor **return** vom Stack weg in Sicherheit gebracht werden, da im Anschluß an **call** Code generiert werden muß, der die Argumentwerte für die Funktion wieder vom Stack entfernt. Dieser Code ist auch dafür verantwortlich, daß ein Resultatwert sich zum Schluß wieder auf dem Stack befindet.

Wir zeigen wieder nebeneinander, wie die Grammatik erweitert wird, und welche Funktionen von den neuen Aktionen aufgerufen werden. Der vollständige Modul für Code-Generierung befindet sich im Abschnitt 6 im Anhang. Als erstes betrachten wir die Code-Generierung für arithmetische Ausdrücke:

```
binary
        : binary '+' binary
                { gen_alu(ALU_ADD, "+"); }
```

Wie fast alle Funktionen für Code-Generierung besteht auch **gen_alu()** nur aus einem Aufruf der Büchereifunktion **printf()**:

```
gen_alu(mod, comment)
        char * mod;             /* ALU Operationsname */
        char * comment;         /* Kommentar */
{
        printf("\t%s\t%s\t\t; %s\n", OP_ALU, mod, comment);
}
```

alu ist die Klasse aller arithmetischen Operationen; jede solche Operation entfernt zwei Werte vom Stack, verknüpft sie, und legt das Resultat wieder auf dem Stack ab. Damit das funktioniert, müssen sich die Operanden jeder arithmetischen Operation auf dem Stack befinden. Wir müssen also zuerst Konstanten und die Werte von Variablen auf den Stack bringen, und zwar in der Reihenfolge, in der sie in einem Ausdruck vorkommen:

```
binary
        : Constant
                { gen_li($1); }
```

s_lookup() hat bekanntlich für **Constant** den zugehörigen Text dynamisch gespeichert, und einen entsprechenden Zeigerwert auf dem Wert-Stack deponiert. Mit diesem Text können wir jetzt einen **load-immediate**-Maschinenbefehl generieren; die Repräsentierung einer **Constant** wird dadurch völlig in der lexikalischen Analyse festgelegt.

```
gen_li(const)
        char * const;           /* Constant: Wert */
{
        printf("\t%s\t%s,%s\n", OP_LOAD, MOD_IMMED, const);
}
```

Um den Wert einer Variablen auf den Stack bringen zu können müssen wir die Symboltabelle konsultieren:

```
%{
#define OFFSET(x)       ( ((struct symtab *) x) -> s_offset )
#define NAME(x)         ( ((struct symtab *) x) -> s_name )
%}
%%

binary
        : Identifier
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
```

gen_mod() konstruiert aus **s_blknum** den richtigen Bereichsnamen, **OFFSET()** und **NAME()** beschaffen die notwendigen Komponenten aus dem Eintrag in der Symboltabelle, und **gen()** schließlich fügt die verschiedenen Felder in einen Maschinenbefehl zusammen:

```
char * gen_mod(symbol)
        struct symtab * symbol;
{
        switch (symbol->s_blknum) {
        case 1:
                return MOD_GLOBAL;
        case 2:
                return MOD_PARAM;
        }
        return MOD_LOCAL;
}

gen(op, mod, val, comment)
        char * op;              /* Befehlsname */
        char * mod;             /* Bereichsname */
        int val;                /* Offset */
        char * comment;         /* Kommentar */
{
        printf("\t%s\t%s,%d\t\t; %s\n", op, mod, val, comment);
}
```

Bei einer Zuweisung müssen wir Code generieren, damit der gewünschte Wert berechnet wird. Eine einfache Zuweisung wird dann mit einem **store**-Maschinenbefehl realisiert. Bei einer Zuweisung mit arithmetischer Verknüpfung, also zum Beispiel bei `+=`, muß zuerst ein **load**-Befehl generiert werden, dann folgt der Code für die rechte Seite der Zuweisung, und schließlich ein entsprechender **alu**-Befehl; erst dann kann ein **store**-Befehl die eigentliche Zuweisung vornehmen. **store** entfernt den zugewiesenen Wert *nicht* vom Stack; damit funktioniert auch die Zuweisung als Operator innerhalb eines arithmetischen Ausdrucks korrekt.

```
binary
        : Identifier '=' binary
                { chk_var($1);
                  gen(OP_STORE, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
```

```
        | Identifier PE
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
          binary
                { gen_alu(ALU_ADD, "+");
                  gen(OP_STORE, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
```

'Glücklicherweise' verfügen wir über spezielle Maschinenbefehle für Inkrement- und Dekrementoperationen, die sowohl den Speicher modifizieren, als auch einen Resultatwert auf dem Stack ablegen. Als Beispiel zeigen wir hier die Inkrementoperation:

```
binary
        : PP Identifier
                { chk_var($2);
                  gen(OP_INC, gen_mod($2), OFFSET($2),
                        NAME($2));
                }
```

Der Kommaoperator sorgt in C für die sequentielle Bewertung von zwei Ausdrücken: der linke Operand wird zuerst bewertet, sein Wert bleibt jedoch unberücksichtigt; anschließend wird der rechte Operand des Kommaoperators bewertet, dieser Wert gilt als Resultat. Damit er unberücksichtigt bleibt, muß der Wert des linken Operanden mit **pop** vom Stack entfernt werden:

```
expression
        : binary
        | expression ','
                { gen_pr(OP_POP, "abraeumen"); }
          binary
                { yyerrok; }
```

Während der Reduktion des zweiten **binary** wird Code generiert, der dann den endgültigen Resultatwert auf dem Stack hinterläßt. Da **pop** und **return** gleiche Befehlsformate besitzen, dient **gen_pr()** später auch dazu, **return**-Maschinenbefehle zu generieren.

```
gen_pr(op, comment)
        char * op;              /* Befehlsname */
        char * comment;         /* Kommentar */
{
        printf("\t%s\t\t\t; %s\n", op, comment);
}
```

Wir können jetzt einige Anweisungen untersuchen. Für die leere Anweisung muß kein Code generiert werden. **compound_statement** wird automatisch als Folge seiner abhängigen Anweisungen codiert. Wird **expression** als Anweisung verwendet, codieren wir wie bei einem Kommaoperator: der Resultatwert muß am Schluß der Anweisung vom Stack entfernt werden.

```
statement
        : sc
        | compound_statement
        | expression sc
                { gen_pr(OP_POP, "Stack abraeumen"); }
```

Kontrollstrukturen werden mit Hilfe von Sprungbefehlen implementiert. Der einfachste Fall ist dabei die **if**-Anweisung:

```
%type   <y_lab> if_prefix
%%
if_prefix
        : IF '(' expression rp
                { $$ = gen_jump(OP_JUMPZ, new_label(), "IF"); }.
```

Für **expression** wird ganz normal Code generiert; dadurch verbleibt auf dem Stack ein Wert. Der **jumpz**-Maschinenbefehl sorgt für einen Sprung genau dann, wenn dieser Wert Null ist; der Wert wird auf alle Fälle durch **jumpz** vom Stack entfernt.

```
int gen_jump(op, label, comment)
        char * op;              /* Befehlsname */
        int label;              /* Sprungziel */
        char * comment;         /* Kommentar */
{
        printf("\t%s\t%s\t\t; %s\n", op, format_label(label),
                comment);
        return label;
}
```

new_label() ist eine Funktion, die bei jedem Aufruf eine neue, eindeutige Marke generieren muß. Eindeutige Marken generieren wir, indem wir eine in **new_label()** mit **static** definierte Integer-Variable hochzählen; der resultierende Wert kann auf dem *yacc*-Wert-Stack weitergereicht werden (und wird von **gen_jump()** zu diesem Zweck als Funktionswert geliefert). Eine einzige Funktion **format_label()** dient dazu, diese eindeutigen Integer-Werte in verschiedenem Kontext für den Assembler als Marken akzeptabel darzustellen.

```
int new_label()
{       static int next_label = 0;

        return ++next_label;
}
```

```
#define LABEL   "$$%d"

static char * format_label(label)
        int label;
{       static char buffer[sizeof LABEL + 2];

        sprintf(buffer, LABEL, label);
        return buffer;
}
```

if_prefix sorgte für die Bewertung der **if**-Bedingung; außerdem wurde die Entscheidung als Sprung zu einer - noch undefinierten - Marke implementiert, die **if_prefix** auf dem *yacc*-Wert-Stack hinterlassen hat. Der Sprung dient zur Umgehung des **then**-Teils der **if**-Anweisung, wenn die Bedingung nicht erfüllt ist. Die nötige Marke müssen wir noch im Anschluß an die abhängige Anweisung definieren:

```
statement
        : if_prefix statement
                { gen_label($1); }
```

gen_label() definiert eine Marke mit Hilfe einer **equ**-Assembler-Anweisung:

```
int gen_label(label)
        int label;
{
        printf("%s\tequ\t*\n", format_label(label));
        return label;
}
```

Hat die **if**-Anweisung einen **else**-Teil, so müssen wir die Marke am Anfang des **else**-Teils definieren. Zuvor muß aber ein unbedingter Sprung codiert werden, der vom Ende des **then**-Teils direkt zum Ende der **if**-Anweisung führt. Auch hier benützen wir den von *yacc* unterhaltenen Wert-Stack um die undefinierten Marken an den Punkt weiterzureichen, an dem sie definiert werden können:

```
statement
        : if_prefix statement ELSE
                { $<y_lab>$ = gen_jump(OP_JUMP, new_label(),
                        "ueber ELSE hinweg");
                  gen_label($1);
                }
          statement
                { gen_label($<y_lab>4); }
```

Fehler im Quellprogramm können dazu führen, daß fehlerhaft codiert wird. Wir müssen zum Beispiel in folgendem Fall einen Sprungbefehl generieren:

```
if_prefix
        : IF error
                { $$ = gen_jump(OP_JUMPZ, new_label(), "IF"); }
```

Befindet sich jetzt kein geeigneter Wert auf dem Stack, verursacht **jumpz** einen Laufzeitfehler. Generieren wir in dieser Situation aber keinen Sprungbefehl, sondern legen nur den von **new_label()** erzeugten Wert direkt auf dem Wert-Stack ab, kann der Laufzeit-Stack langfristig überlaufen!

while wird mit der gleichen Technik implementiert:

```
%type   <y_lab> loop_prefix
%%

loop_prefix
        : WHILE '('
                { $<y_lab>$ = gen_label(new_label());
                  push_continue($<y_lab>$);
                }
          expression rp
                { $$ = $<y_lab>3; }
        | WHILE error
                { $$ = gen_label(new_label());
                  push_continue($$);
                }
```

loop_prefix definiert zuerst eine neue Marke als Fortsetzungspunkt der Schleife. Anschließend wird Code für die Bedingung generiert. Die (definierte) Marke wird auf dem *yacc*-Wert-Stack hinterlegt.

Der Sprungbefehl, der die Bedingung prüft, wird auf der **statement**-Ebene codiert. Damit kann das Sprungziel, eine zweite, neue, undefinierte Marke in einem weiteren Element auf dem *yacc*-Wert-Stack weitergereicht werden:

```
statement
        : loop_prefix
                { $<y_lab>$ = gen_jump(OP_JUMPZ, new_label(),
                                "WHILE");
                  push_break($<y_lab>$);
                }
          statement
                { gen_jump(OP_JUMP, $1, "zurueck zu WHILE");
                  gen_label($<y_lab>2);
                  pop_break();
                  pop_continue();
                }
```

Anschließend kann Code für die abhängige Anweisung generiert werden. Darauf folgt ein unbedingter Sprung zum Fortsetzungspunkt. Zum Schluß muß die Marke generiert werden, die als Sprungziel dient, wenn die **while**-Schleife verlassen werden soll.

Diese Implementierung ist nicht optimal: wenn wir den Code für die Bedingung erst *nach* der abhängigen Anweisung einfügen würden, könnten wir die Schleife mit einem bedingten Sprung wiederholen und damit einen Sprungbefehl pro Schleifendurchgang einsparen. Dies bedingt jedoch, daß wir den Code für **expression** irgendwo zwischenzeitlich aufbewahren.

kleinC hat **break**- und **continue**-Anweisungen, die innerhalb einer **while**-Schleife als Sprünge zum Abbruch- oder Fortsetzungspunkt der Schleife realisiert werden. Für diese unbedingten Sprünge müssen wir entsprechende Marken bereitstellen. Da **while**-Anweisungen verschachtelt werden können, müssen diese Marken auf einem Stack stehen. Da in C eine **switch**-Anweisung zwar ein neues Ziel für **break** aber nicht für **continue** definiert, verwenden wir zwei Stacks, obgleich *kleinC* prinzipiell mit einem solchen Stack übersetzt werden könnte.

Das Management dieser Stacks wurde bei **loop_prefix** und **while** bereits berücksichtigt: bei **loop_prefix** werden Marken auf die Stacks gebracht, im Zusammenhang mit **while** werden sie wieder entfernt. Wenn wir die Existenz geeigneter **push()**- und **pop()**-Funktionen zum Stack-Management voraussetzen, können die **break**- und **continue**-Stacks folgendermaßen manipuliert werden:

```
push_break(label)
        int label;
{
        b_top = push(b_top, label);
}

push_continue(label)
        int label;
{
        c_top = push(c_top, label);
}

pop_break()
{
        b_top = pop(b_top);
}

pop_continue()
{
        c_top = pop(c_top);
}
```

Setzen wir auch noch die Existenz einer Funktion **top()** voraus, mit der ein Stack inspiziert werden kann, so sind **break** und **continue** sehr leicht zu realisieren:

```
statement
        : BREAK sc
                { gen_break(); }
        | CONTINUE sc
                { gen_continue(); }

gen_break()
{
        gen_jump(OP_JUMP, top(b_top), "BREAK");
}

gen_continue()
{
        gen_jump(OP_JUMP, top(c_top), "CONTINUE");
}
```

Auch das Stack-Management selbst ist nicht besonders schwierig. Wir definieren folgende Struktur, um eine interne Marke zu speichern und die gespeicherten Marken als Stack zu verketten:

```
static struct bc_stack {
        int bc_label;           /* Marke (von new_label()) */
        struct bc_stack * bc_next;
        } * b_top,              /* Beginn BREAK-Stack */
          * c_top;              /* Beginn CONTINUE-Stack */
```

b_top zeigt auf das oberste Element im **break**-Stack, **c_top** entsprechend im **continue**-Stack. Beide Zeiger sind Nullzeiger, wenn die Stacks leer sind, das heißt, wenn **break**- und **continue**-Anweisungen (außerhalb von **while**-Schleifen) nicht erlaubt sind.

push() erhält als Argumente einen Zeiger auf einen Stack sowie eine Marke, die auf dem Stack abgelegt werden soll. **push()** liefert als Resultat einen Zeiger auf ein neues Stack-Element, das die Marke enthält, und mit dem der restliche Stack verkettet ist. Dieser Resultatwert muß noch an den Zeiger zugewiesen werden, der an **push()** als Argument übergeben wurde, erst dadurch wird die **push()**-Operation vervollständigt.

```
static struct bc_stack * push(stack, label)
        struct bc_stack * stack;
        int label;
{       struct bc_stack * new_entry = (struct bc_stack *)
                calloc(1, sizeof(struct bc_stack));
```

```
        if (new_entry)
        {       new_entry->bc_next = stack;
                new_entry->bc_label = label;
                return new_entry;
        }
        fatal("Kein Platz mehr fuer Schleifen.");
        /*NOTREACHED*/
}
```

Analog entfernt **pop()** das oberste Element von einem Stack und liefert die Kette der restlichen Elemente; auch dieser Zeigerwert muß an den Zeiger zugewiesen werden, der als Argument an **pop()** übergeben wurde.

```
static struct bc_stack * pop(stack)
        struct bc_stack * stack;
{       struct bc_stack * old_entry;

        if (stack)
        {       old_entry = stack;
                stack = old_entry->bc_next;
                cfree(old_entry);
                return stack;
        }
        bug("break/continue Stack ist leer");
        /*NOTREACHED*/
}
```

In **top()** findet die Semantikprüfung für **break** und **continue** statt:

```
static int top(stack)
        struct bc_stack * stack;
{
        if (! stack)
        {       error("keine Schleife offen");
                return 0;
        }
        else
                return stack->bc_label;
}
```

Nur noch eine Operation muß implementiert werden: wir müssen noch Code für einen Funktionsaufruf und für Prolog und Epilog einer Funktion generieren. Eine Funktion wird mit einem **call**-Maschinenbefehl aufgerufen:

```
binary
        : Identifier '('
                { chk_func($1); }
          optional_argument_list rp
                { gen_call($1,$4); }
```

Da die Liste der Argumente zuerst reduziert wird, findet die Funktion die Argumentwerte auf dem Stack vor. Nach Rückkehr vom Funktionsaufruf müssen wir diese Werte vom Stack entfernen. Es ist nicht einfach, ein Funktionsresultat auf dem Stack zu übergeben. Wir übergeben deshalb ein Funktionsresultat ganz am Anfang des globalen Bereichs; **gen_call()** muß diesen Wert dann noch auf den Stack bringen, denn dort soll ja im weiteren Verlauf der Funktionswert die Argumentwerte ersetzen:

```
gen_call(symbol, count)
        struct symtab * symbol; /* Funktion */
        int count;              /* Anzahl Argumente */
{
        chk_parm(symbol, count);
        printf("\t%s\t%d,%s\n", OP_CALL, count, symbol->s_name);
        while (count-- > 0)
                gen_pr(OP_POP, "Argument entfernen");
        gen(OP_LOAD, MOD_GLOBAL, 0, "Resultat auf Stack");
}
```

Es gibt einen **return**-Maschinenbefehl, der für den Rücksprung zum zugehörigen **call**-Befehl sorgt. **call** deponiert die Rücksprungadresse auf dem Stack, **return** entfernt sie. Damit können diese Befehle rekursiv verwendet werden. Soll ein Resultatwert übergeben werden, so befindet er sich auf dem Stack. Bevor der **return**-Befehl codiert werden kann, müssen wir diesen Resultatwert noch an das Resultatwort am Anfang des globalen Bereichs zuweisen und vom Stack entfernen:

```
statement
        : RETURN sc
                { gen_pr(OP_RETURN, "RETURN"); }
        | RETURN expression sc
                { gen(OP_STORE, MOD_GLOBAL, 0,
                        "Resultat speichern");
                  gen_pr(OP_RETURN, "RETURN");
                }
```

call legt die Rücksprungadresse auf dem Stack ab. Wird eine Funktion aufgerufen, so muß die Adresse des alten Prozedursegments auf dem Stack abgelegt werden. Das Register, das den Anfang des Prozedursegments markiert, muß auf den Anfang des neuen Prozedursegments, also auf die Stack-Spitze, eingestellt werden, und auf dem Stack muß genügend Platz für das neue Prozedursegment bereitgestellt werden. Dies alles erledigt der **entry**-Maschinenbefehl, der die Größe des neuen Prozedursegments als Parameter benötigt:

```
function_definition
        : Identifier '('
                { make_func($1);
                  blk_push();
                }
          optional_parameter_list rp
          parameter_declarations
                { chk_parm($1, parm_default($4));
                  all_parm($4);
                  l_max = 0;
                  $<y_lab>$ = gen_entry($1);
                }
          compound_statement
                { all_func($1);
                  gen_pr(OP_RETURN, "Funktionsende");
                  fix_entry($1, $<y_lab>7);
                }
```

Leider kennen wir die Größe des Prozedursegments noch nicht, wenn wir eine neue Funktion übersetzen. Als Parameter für **entry** verwenden wir daher eine neue, interne Marke, die wir auf dem *yacc*-Wert-Stack weiterreichen:

```
int gen_entry(symbol)
        struct symtab * symbol; /* Funktion */
{       int label = new_label();

        printf("%s\t", symbol->s_name);
        printf("%s\t%s\n", OP_ENTRY, format_label(label));
        return label;
}
```

Im Anschluß an die Übersetzung der Funktion können wir dann diese Marke entsprechend definieren:

```
fix_entry(symbol, label)
        struct symtab * symbol; /* Funktion */
        int label;
{       extern int l_max;       /* Groesse, lokaler Bereich */

        printf("%s\tequ\t%d\t\t; %s\n", format_label(label),
                l_max, symbol->s_name);
}
```

Wir sind fertig. Wenn wir das ganze Quellprogramm übersetzt haben, kennen wir die Größe des globalen Bereichs, und wir verlangen, daß die Ausführung mit dem Aufruf einer **main()**-Funktion beginnen soll, die sich irgendwo in der Quelle befinden muß. Diese Informationen reichen wir mit Hilfe einer **end**-Assembler-Anweisung weiter:

```
program
        :       { init(); }
          definitions
                { end_program(); }

end_program()
{       extern int g_offset;    /* Groesse, globaler Bereich */

        all_program();          /* globale Variablen anlegen */
        printf("\tend\t%d,main\n", g_offset);
}
```

Als Beispiel zur Code-Generierung betrachten wir folgende Implementierung von Euklid's Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier positiver Zahlen:

```
/*
 *      Euklid's Algorithmus
 */

main()
{       int a,b;

        a = 36;
        b = 54;

        while (a != b)
                if (a > b)
                        a -= b;
                else
                        b -= a;
}
```

Folgender Assembler-Text wird generiert:

```
main    entry   $$1
        load    con,36
        store   lcl,0           ; a
        pop                     ; Stack abraeumen
        load    con,54
        store   lcl,1           ; b
        pop                     ; Stack abraeumen
$$2     equ     *
        load    lcl,0           ; a
        load    lcl,1           ; b
        alu     !=              ; !=
        jumpz   $$3             ; WHILE
        load    lcl,0           ; a
```

```
        load    lcl,1           ; b
        alu     >               ; >
        jumpz   $$4             ; IF
        load    lcl,0           ; a
        load    lcl,1           ; b
        alu     -               ; -
        store   lcl,0           ; a
        pop                     ; Stack abraeumen
        jump    $$5             ; ueber ELSE hinweg
$$4     equ     *
        load    lcl,1           ; b
        load    lcl,0           ; a
        alu     -               ; -
        store   lcl,1           ; b
        pop                     ; Stack abraeumen
$$5     equ     *
        jump    $$2             ; zurueck zu WHILE
$$3     equ     *
        return                  ; Funktionsende
$$1     equ     2               ; main
        end     1,main
```

7.3 Aufgaben

1. Implementieren Sie *kleinC* auf einer existenten Maschine. Eine Lösungsmöglichkeit besteht darin, die in diesem Kapitel verwendeten Maschinenbefehle mit Makros auf der neuen Maschine nachzubilden. Der Code-Generator kann auch direkt modifiziert werden, wobei man wohl eine Stack-Architektur als Ziel beibehält. Eine Implementierung, die möglichst optimal an eine ganz andere Architektur, wie zum Beispiel die einer Registermaschine, angepaßt ist, ist entscheidend schwieriger zu realisieren. ✏

2. Im letzten Beispiel zeigt sich, daß der generierte Assembler-Text nicht immer sehr effizient ist. Beispielsweise wird folgende Befehlsfolge generiert:

```
        jump    $$5             ; ueber ELSE hinweg
        ...
$$5     equ     *
        jump    $$2             ; zurueck zu WHILE
```

Sprünge können nur zu Marken erfolgen. Geht eine solche Marke direkt einem **jump**-Befehl voraus, so kann die Befehlsfolge optimiert werden, indem man die Marke mit dem Ziel des zweiten Sprungbefehls durch **equ** gleichsetzt. Bei diesem Beispiel ergibt sich:

```
$$5     equ     $$2
```

an Stelle der früheren **equ**-Assembler-Anweisung. Wie erreicht man dies? Hinweis: man könnte einen Postprozessor einführen, der die Ausgabe vom Compiler optimiert, bevor sie an den Assembler übergeben wird. Kann man die Optimierung auch im Ein-Pass-Compiler selbst durchführen? ✏

3. Ebenfalls ineffizient sind Befehlsfolgen, die nie ausgeführt werden *können*. Wenn etwa eine **return**-Anweisung direkt vor dem Ende einer Funktion steht, generiert unser *kleinC*-Compiler unmittelbar nacheinander *zwei* **return**-Maschinenbefehle. Dies ist grundsätzlich bei einer Funktion der Fall, die explizit mit **return** ein Resultat liefert. Wie kann man die zweite **return**-Befehlsfolge unterdrücken? ✏

4. Desweiteren ist eine Serie von aufeinanderfolgenden **load**- und **pop**-Maschinenbefehlen unnötig. Besteht zum Beispiel ein Anweisung in *kleinC* nur aus einem Funktionsaufruf, wird also der Resultatwert gar nicht beachtet, so erzeugt unser Compiler einen **load**-Befehl um das Funktionsresultat wieder auf den Stack zu bringen, und sofort anschließend einen **pop**-Befehl als Abschluß der Anweisung. Wie lassen sich derartige unnötige Befehlsfolgen vermeiden? ✏

Kapitel 8: Simulation

Im letzten Kapitel sahen wir, daß ein Code-Generator relativ leicht konstruiert werden kann, vorausgesetzt die Architektur der Zielmaschine reflektiert die Quellsprache gut genug. Haben wir die *Implementierung* definiert, das heißt, Speichermanagement und Befehlsfolgen für die verschiedenen Aktionen der Quellsprache, dann können wir zum Beispiel durch Simulation einer geeigneten Maschinenarchitektur unser Programmiersystem vervollständigen. In diesem Kapitel betrachten wir, wie man einen Simulator für die Maschine konstruiert, für die wir in den vorhergehenden beiden Kapiteln Code generiert haben. Wir untersuchen außerdem, wie man direkt im (simulierten) Speicher dieser Maschine Code generiert, das heißt, wie man zusammen mit dem Simulator ein sogenanntes *load-and-go* System realisiert. Die Diskussion befaßt sich notwendigerweise mit den Details von *kleinC*. Verschiedene Sprachen sind jedoch ähnlich implementiert worden, zum Beispiel Pascal und Modula-2.

8.1 Ein Maschinensimulator

Ein Simulator für unsere fiktive Maschine ist recht einfach zu konstruieren. Zuerst müssen wir die verschiedenen Maschinenbefehle und Befehlsparameter repräsentieren, am besten wohl als Zahlenwerte. Dies geschieht in der Definitionsdatei *sim.h*, in der alle Namen entsprechend neu vereinbart werden, die zuvor schon in *gen.h* definiert wurden:

```
/*
 *      Befehlsnamen fuer fiktive Maschine
 */

#define OP_ALU    1     /* alu     Arithmetik-Logik-Op  */
#define OP_DEC    2     /* dec     Bereich,Offset       */
#define OP_INC    3     /* inc     Bereich,Offset       */
#define OP_LOAD   4     /* load    Bereich,Offset       */
#define OP_STORE  5     /* store   Bereich,Offset       */
#define OP_POP    6     /* pop                          */
#define OP_JUMPZ  7     /* jumpz   Marke                */
#define OP_JUMP   8     /* jump    Marke                */
#define OP_CALL   9     /* call    Funktionsadresse     */
#define OP_ENTRY  10    /* entry   lokaler-Bereich      */
#define OP_RETURN 11    /* return                       */
```

```
/*
 *      Bereichsnamen
 */

#define MOD_GLOBAL 1    /* globale Variable     */
#define MOD_PARAM  2    /* Parameter            */
#define MOD_LOCAL  3    /* lokale Variable      */
#define MOD_IMMED  4    /* "load": Constant     */

/*
 *      OP_ALU Namen
 */

#define ALU_ADD 1       /* Addition             */
#define ALU_SUB 2       /* Subtraktion          */
#define ALU_MUL 3       /* Multiplikation       */
#define ALU_DIV 4       /* Division             */
#define ALU_MOD 5       /* Rest                 */
#define ALU_LT  6       /* Vergleich:   <       */
#define ALU_GT  7       /*              >       */
#define ALU_LE  8       /*              <=      */
#define ALU_GE  9       /*              >=      */
#define ALU_EQ  10      /*              ==      */
#define ALU_NE  11      /*              !=      */
#define ALU_AND 12      /* Bit-UND              */
#define ALU_OR  13      /* Bit-ODER             */
#define ALU_XOR 14      /* Bit-ODER exklusiv    */
```

Zusätzlich müssen wir Vektoren als Programm- und Datenspeicher für unsere *kleinC*-Maschine definieren. Elemente des Datenspeichers repräsentieren wir als Integer-Werte, aber für den Programmspeicher vereinbaren wir eine Elementstruktur, um den Aufwand beim Decodieren zu reduzieren:

```
struct prog {
        short p_op;     /* Befehlsname */
        short p_mod;    /* Bereich/ALU Name */
        int p_val;      /* Offset o. ae. */
        };

/*
 *      aenderbare Groessen
 */

#define L_PROG  200     /* max. Programmgroesse */
#define L_DATA  100     /* max. fuer Stack und Variablen */
```

```
/*
 *      Ausdehnung eines Vektors
 */

#define DIM(x)  (sizeof x / sizeof x[0])
```

Mindestens die maximale Größe des Programmspeichers und die **prog**-Struktur müssen in der Definitionsdatei vereinbart werden, denn wir benötigen sie nicht nur im Simulator, sondern auch wenn wir dann Code generieren. Im Simulator definieren wir den Datenspeicher und deklarieren den Programmspeicher:

```
#include "sim.h"

/*
 *      Daten- und Programmspeicher
 */

static int data[L_DATA];
extern struct prog prog[];
```

prog[] definieren wir dann im Code-Generator-Modul. Dadurch können wir die Größe der Vektoren jeweils mit dem in der Definitionsdatei definierten **DIM()**-Makro vom C-Compiler aus der Definition der Vektoren ableiten lassen, und wir müssen uns nicht darauf verlassen, daß wirklich die Konstanten aus der Definitionsdatei zur Dimensionierung der Vektoren verwendet wurden.

Als Nächstes definieren wir die Register. Wir verwenden ein 'internes' Register **inst**, das den momentanen Befehl enthält; damit wird die Decodierung vereinfacht. Verschiedene Positionen im Datenspeicher werden mit den Registern **G**, **P**, **L** und **T** markiert: **G** bezeichnet den Anfang des globalen Bereichs, ist also der konstante Indexwert Null. **P** verweist auf den Beginn des aktuellen Parameterbereichs, der dem *activation record* vorausgeht. **L** zeigt auf den lokalen Bereich, also auf den *activation record* selbst. **T** schließlich definiert die Stack-Spitze, das heißt das erste Element im Datenspeicher, das dem lokalen Bereich und allen bereits auf dem Stack abgelegten Werten folgt, und das als nächstes Element verwendet werden kann.

```
/*
 *      Register
 */

static struct prog * inst;  /* -> aktueller Befehl */
#define G       0           /* globaler Bereich */
static int P;               /* aktueller Parameterbereich */
static int L;               /* aktueller lokaler Bereich */
static int T;               /* Stack-Spitze */
```

Es wird sich zeigen, daß gewisse Abkürzungen hier ganz praktisch sind: **TOP** und **NEXT** bezeichnen die obersten zwei Werte auf dem Stack; an **PUSH** werden

wir zuweisen um einen Wert auf den Stack zu bringen; mit der **POP**-Operation entfernen wir das oberste Element vom Stack; **RESULT** steht für das Element am Anfang das Datenspeichers, das wir verwenden um den Resultatwert einer Funktion zu speichern; **MEMORY** schließlich ist die Abkürzung für das Element im Datenspeicher, auf das der aktuelle Maschinenbefehl verweist.

```
/*
 *      Abkuerzungen
 */

#define TOP     data[T-1]    /* rechter Operand: Stack-Spitze */
#define NEXT    data[T-2]    /* linker Operand: unter TOP */
#define PUSH    data[T++]    /* neuer Eintrag auf den Stack */
#define POP     -- T         /* -> Eintrag, vom Stack entfernt */
#define MEMORY  data[address()]
                             /* effektiv adressiertes Wort */
#define RESULT  data[G]      /* Resultatwert einer Funktion */
```

Mit diesen Definitionen erweitern wir die Sprache, in der der Simulator programmiert wird. Wir finden, daß sich unser Programm dadurch besser selbst dokumentiert. Zum Exzeß getrieben, machen solche Spracherweiterungen jedoch ein Programm undurchsichtig und dadurch oft sogar schwieriger zu modifizieren.

MEMORY verwendet eine Funktion **address()** die das Element in **data[]** bestimmen muß, auf das der aktuelle Maschinenbefehl verweist:

```
static int address()    /* effektive Datenadresse */
{       register int ad;

        switch (inst->p_mod) {
        case MOD_GLOBAL:
                ad = G;
                break;
        case MOD_PARAM:
                ad = P;
                break;
        case MOD_LOCAL:
                ad = L;
                break;
        default:
                bug("p_mod ist falsch");
        }
        ad += inst->p_val;
        if (ad < 0 || ad >= T)
                bug("effektive Adresse ist falsch");
        return ad;
}
```

Die effektive Adresse entsteht, indem zum Offset-Wert aus dem Maschinenbefehl die Anfangsadresse des Bereichs addiert wird, auf den die Bereichsangabe im Maschinenbefehl verweist. Die resultierende Adresse darf die aktuelle Größe des Datenspeichers nicht überschreiten.

Wir können jetzt den Befehlszyklus unseres Simulators entwerfen. **simulate()** bezieht die notwendige Startinformation aus der **end**-Anweisung des übersetzten Programms: wenn diese Assembleranweisung generiert wird, kennen wir **pc_limit**, die Größe des übersetzten Programms; **global**, die Größe des globalen Bereichs; und **pc**, die Adresse im Programmspeicher, bei der die Simulation beginnen soll, das heißt, die Anfangsadresse der **main()**-Funktion im übersetzten Programm. Diese Werte können an **simulate()** als Parameter übergeben werden; **pc** dient dann zusätzlich als Programmzähler der simulierten Maschine.

```
simulate(pc_limit, global, pc)
        int pc_limit, global, pc;
{
        /* initialisieren */

        printf("\nAusfuehrung beginnt...\n\n");

        for (;;)
        {       /* fetch */
                if (pc < 0 || pc >= pc_limit)
                        bug("pc nicht im Programm");
                inst = &prog[pc++];

                /* Operation decodieren */
                switch (inst->p_op) {
                default:
                        printf("%d:\thalt\n", inst-prog);
                        return;

                /* andere Maschinenbefehle */
                }
        }
}
```

Wir wollen die Initialisierung der Register **P**, **L** und **T** zurückstellen. Dieses Problem können wir erst lösen, wenn wir die Befehlsfolge für einen Funktionsaufruf implementiert haben: wir müssen diese Register und den Stack so initialisieren, daß ein **return**-Maschinenbefehl in der **main()**-Funktion unsere Simulation anhält.

Betrachten wir nun die verschiedenen Maschinenbefehle in der Reihenfolge, in der sie im Kapitel 7 eingeführt wurden.

alu ist einfach: wir decodieren die gewünschte arithmetische oder logische Operation mit einer zweiten **switch**-Anweisung und bilden dadurch den **alu**-Namen

auf den entsprechenden C-Operator ab. Um das Beispiel möglichst einfach zu halten, treffen wir keine Vorkehrungen zur Erkennung von arithmetischen Problemen wie zum Beispiel Division durch Null. Wir versuchen jedoch festzustellen, ob sich wirklich Operanden auf dem Stack befinden. Dies ist ganz bestimmt nicht der Fall, wenn zwischen dem Anfang des lokalen Bereichs (in **L**) und der Stack-Spitze (in **T**) nicht wenigstens Platz für zwei Operanden ist. Wir zeigen hier nur ein paar typische Beispiele:

```
case OP_ALU:
        if (T <= L+1)
                bug("Simulator-Stack leer");
        switch (inst->p_mod) {
        default:
                bug("illegaler ALU Befehl");
        case ALU_ADD:
                NEXT += TOP;
                break;
        case ALU_LT:
                NEXT = NEXT < TOP;
                break;
        }
        POP;
        break;
```

Wenn die Übersetzung ohne Fehler verläuft, dann kann der Stack im entscheidenden Fall nicht leer sein. **Simulator-Stack leer** ist dann wirklich ein **bug**, also ein Problem, das nicht auftreten darf. Falls das *kleinC*-Programm Fehler enthielt, kann der Stack auch aufgrund fehlerhafter Code-Generierung leer sein. Da unser Test ohnedies nicht sehr genau sein kann (der lokale Bereich selbst ist selten leer), sehen wir die Situation als **bug** an.

Beim **load**-Maschinenbefehl müssen wir unterscheiden, ob wir den Wert einer Variablen aus **MEMORY** oder den Wert einer **Constant** direkt aus dem Befehl holen. In jedem Fall prüfen wir, daß der Stack den verfügbaren Datenspeicher nicht überschreitet, wenn wir den Wert auf den Stack bringen:

```
case OP_LOAD:
        if (T >= DIM(data))
                fatal("Zu viele Werte.");
        if (inst->p_mod == MOD_IMMED)
                PUSH = inst->p_val;
        else
                PUSH = MEMORY;
        break;
```

store kopiert(!) einen Wert vom Stack in den Datenspeicher. Wir versuchen wieder sicherzustellen, daß auch ein Wert auf dem Stack vorhanden ist.

```
case OP_STORE:
        if (T <= L)
                bug("Simulator Stack ist leer");
        printf("%d:\tstore\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
                inst->p_val, MEMORY = TOP);
        break;
```

store bietet sich an um den Effekt der Simulation vorzuführen: wir zeigen im Protokoll an, daß ein **store**-Maschinenbefehl ausgeführt wird, und wir geben den betroffenen Wert aus.

inc und **dec** ändern nicht nur den Wert einer Variablen im Datenspeicher, sondern kopieren das Resultat auch auf den Stack, der dabei wieder nicht zu viel Platz beanspruchen darf:

```
case OP_INC:
        if (T >= DIM(data))
                fatal("Zu viele Werte.");
        printf("%d:\tinc\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
                inst->p_val, PUSH = ++ MEMORY);
        break;

case OP_DEC:
        if (T >= DIM(data))
                fatal("Zu viele Werte.");
        printf("%d:\tdec\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
                inst->p_val, PUSH = -- MEMORY);
        break;
```

Auch diese Befehle erscheinen im Ablaufprotokoll, damit alle Änderungen von Variablenwerten verfolgt werden können.

pop entfernt einen Wert vom Stack, wenigstens falls es plausibel ist, daß ein solcher Wert existiert:

```
case OP_POP:
        if (T <= L)
                bug("Simulator Stack ist leer");
        POP;
        break;
```

jump verändert den Programmzähler (**pc**) der Simulation. Im weiteren Verlauf der Simulationsschleife wird der neue Wert als Adresse des nächsten Befehls verwendet. Die Existenz der Adresse wird beim Decodieren des Befehls verifiziert.

```
case OP_JUMP:
        printf("%d:\tjump\t%d\n", inst-prog,
                inst->p_val);
        pc = inst->p_val;
        break;
```

jumpz entfernt einen Wert vom Stack und überprüft ihn. Ist der Wert Null, so wird der Programmzähler verändert; dadurch findet dann ein Sprung statt:

```
case OP_JUMPZ:
        if (T <= L)
                bug("Simulator Stack ist leer");
        if (data[POP] == 0)
        {       printf("%d:\tjumpz\t%d\n",
                        inst-prog, inst->p_val);
                pc = inst->p_val;
        }
        break;
```

Wenn ein Sprung stattfindet, geben wir den Sprungbefehl ins Ablaufprotokoll aus, damit der Programmfluß verfolgt werden kann.

call bringt den Wert des Programmzählers auf den Stack. Da beim Decodieren des Befehls der Programmzähler schon auf den nachfolgenden Befehl zeigt, ist dieser Wert die korrekte **return**-Adresse. **call** legt auch die Anfangsadresse des aktuellen Parameterbereichs auf dem Stack ab, und sorgt dafür, daß Register **P** auf den neuen Parameterbereich auf dem Stack zeigt. Die notwendige Adresse läßt sich leicht berechnen, da die Anzahl der übergebenen Argumente im **call**-Befehl steht, und da die Argumentwerte unmittelbar vor **call** auf dem Stack abgelegt wurden. Anschließend findet ein Sprung zur gewünschten Funktion statt. Auch **call** wird ins Ablaufprotokoll eingetragen:

```
case OP_CALL:
        printf("%d:\tcall\t%d\n", inst-prog,
                inst->p_val);
        PUSH = pc;
        pc = inst->p_val;
        PUSH = P;
        P = T - 2 - inst->p_mod;
        break;
```

Am Anfang der neuen Funktion muß die Adresse des alten lokalen Bereichs (in **L**) ebenfalls auf dem Stack abgelegt werden. Der neue lokale Bereich wird auf dem Stack reserviert, indem **L** aus **T** die Adresse der aktuellen Stack-Spitze übernimmt, und **T** entsprechend der Größe des neuen lokalen Bereichs verändert wird. Dies alles ist die Aufgabe des **entry**-Maschinenbefehls:

```
case OP_ENTRY:
        PUSH = L;
        L = T;
        T += inst->p_val;
        if (T >= DIM(data))
                fatal("Zu viele Werte.");
        break;
```

Bei **return** müssen wir den aktuellen lokalen Bereich vom Stack eliminieren, indem wir **T** auf die Anfangsadresse des Bereichs (in **L**) setzen. Die Adressen des vorhergehenden lokalen Bereichs und des vorhergehenden Parameterbereichs können dann auf dem Stack gefunden werden. Unmittelbar darunter befindet sich die Rücksprungadresse. Alle drei Werte müssen ebenfalls vom Stack entfernt werden:

```
case OP_RETURN:
        if (T < L)
                bug("Simulator Stack ist leer");
        T = L;
        L = data[POP];
        P = data[POP];
        pc = data[POP];
        printf("%d:\treturn\t%d to %d\n",
                inst-prog, RESULT, pc);
        break;
```

Ein etwaiger Resultatwert wird später aus **RESULT** auf den Stack gebracht; wir können ihn zusammen mit **return** auf alle Fälle im Ablaufprotokoll zeigen.

Funktionsaufruf und **return** könnte man noch vereinfachen. Wir müssen die Argumentwerte nicht unbedingt mit Hilfe von **pop**-Maschinenbefehlen entfernen; die ursprüngliche Position der Stack-Spitze könnte als Anfang des Parameterbereichs wiederhergestellt werden. Darüberhinaus wird das **RESULT**-Element nicht benötigt; da die ursprüngliche Stack-Spitze berechenbar ist, kann **return** jeweils dort einen Resultatwert ablegen. Diese Änderungen verkürzen zwar den generierten Code, sie machen jedoch den Simulator schwieriger zu verstehen; in jedem Fall ist das ganze Vorgehen nicht gerade realistisch - **return**-Maschinenbefehle auf wirklicher Hardware sind erheblich primitiver!

Der Simulator ist damit vollständig und muß nur noch initialisiert werden. Wir sahen gerade, daß **return** auf dem Stack einen Programmzähler und die Adresse eines Parameterbereichs (von **call**) und die Adresse eines lokalen Bereichs (von **entry**) erwartet. Wir müssen durch Initialisierung einen **call**-Befehl für **main()** simulieren; erfolgt dann **return**, soll der Simulator anhalten.

Der Simulator wird dadurch angehalten, daß wir Null als Maschinenbefehl am Anfang des Programmspeichers ablegen. Alle undefinierten Sprünge oder Funktionsaufrufe halten damit den Simulator an, denn sie würden Adresse Null, also den

Anfang des Programmspeichers, als Zieladresse verwenden. Zusätzlich definieren wir Null als **return**-Adresse auf dem Stack, bevor wir die Simulation bei dem **entry**-Maschinenbefehl in **main()** beginnen; dadurch hält auch ein **return**-Maschinenbefehl in **main()** den Simulator an. Anfangswerte für **P** und **L** werden nicht benötigt, da diese Werte nicht verwendet werden. Wir können auch den Wert von **P** undefiniert lassen, den der fiktive **call**-Maschinenbefehl für **main()** auf den Stack gebracht hat:

```
if (global >= DIM(data))
   fatal(
      "Nicht genuegend Platz fuer globale Variablen.");
T = global + 2;
```

Der Programmspeicher ist **static** definiert und damit auf Null initialisiert.

8.2 Code-Generierung im Speicher

Wenn wir einen Simulator verwenden, bietet es sich an, daß der Compiler direkt im simulierten Speicher Code generiert. Wurde ein Programm - erfolgreich - übersetzt, so kann es dann sofort 'ausgeführt' werden.

Code-Generierung im Speicher ist identisch zur Ausgabe von Assembler-Text, wenigstens was die Befehlsfolgen angeht. Die Reparatur von Vorwärtsverweisen kann man jedoch nicht mehr einem Zwei-Pass-Assembler überlassen; dies muß jetzt der Compiler selbst erledigen, bevor die Simulation beginnen kann. In Abhängigkeit von der Darstellung der Befehle im (simulierten) Speicher ist möglicherweise auch sehr viel Bit-Manipulation bei der Code-Generierung nötig; speziell in diesem Bereich vermeidet man durch die Generierung von Assembler-Text viel Ärger, und man macht den Compiler sehr viel leichter portierbar.

In diesem Abschnitt betrachten wir Funktionen zur Code-Generierung im Speicher, die zusammen mit dem Simulator aus dem vorhergehenden Abschnitt verwendet werden. Es zeigt sich, daß die neuen Funktionen nach Namen und Aufrufkonventionen identisch zu den in Kapitel 7 betrachteten Funktionen sind. Eine neue Definitionsdatei, *sim.h*, definiert eine neue Repräsentierung der Befehlsnamen; davon abgesehen bleibt jedoch der Parser absolut unverändert!

Wir definieren zuerst den Programmspeicher und konstruieren eine Funktion, die einen Befehl im Programmspeicher ablegt. Wie ein Assembler braucht auch ein Code-Generator einen privaten Programmzähler **pc**, der auf das nächste verfügbare Element im Programmspeicher verweist. Zum Schluß wird dann **pc** als **pc_limit** an den Simulator übergeben, da der abschließende Wert von **pc** gerade die Größe des belegten Teils im Programmspeicher definiert.

```
#include "sim.h"
#include "symtab.h"

/*
 *      Programmspeicher
 */

struct prog prog[L_PROG];
static int pc = 1;                  /* aktueller Programmzaehler */
                                    /* HALT (0) ist bei Adresse 0 */

/*
 *      einen Befehl im Programmspeicher ablegen
 */

int gen(op, mod, val, comment)
        int op;                     /* Befehlsname */
        int mod;                    /* Bereich/ALU Name */
        int val;                    /* Offset o.ae. */
        char * comment;             /* Kommentar */
{
        if (pc >= DIM(prog))
                fatal("Kein Platz mehr im Programmspeicher.");
        prog[pc].p_op = op;
        prog[pc].p_mod = mod;
        prog[pc].p_val = val;
        printf("%d:\t%d\t%d,%d\t; %s\n",
                pc, op, mod, val, comment);
        return pc ++;
}
```

Der Programmzähler wird mit der Adresse **1** initialisiert. Bei Adresse **0** enthält der Programmspeicher den Wert Null. Wie im vorhergehenden Abschnitt ausgeführt wurde, führen undefinierte Funktionen und ein **return**-Maschinenbefehl in **main()** zu einem Sprung zu Adresse Null, und damit dazu, daß der Simulator anhält.

gen() ist die einzige Funktion, die neue Maschinenbefehle im Programmspeicher ablegt. Diese Funktion bietet sich daher an um die Code-Generierung durch entsprechende Einträge im Protokoll verfolgen zu lassen. **gen()** liefert als Resultat den Wert des Programmzählers, bei dem der Befehl generiert wurde. Wie wir noch sehen werden, benötigen einige andere Funktionen der Code-Generierung diesen Wert. Auf die Aufrufe von **gen()** innerhalb von Aktionen im Parser hat dies keinen Einfluß.

Bereichsnamen werden genauso berechnet wie vorher, nur liefert **gen_mod()** jetzt eine Darstellung als Integer-Wert:

```
int gen_mod(symbol)
        struct symtab * symbol;
{
        switch (symbol->s_blknum) {
        case 1:
                return MOD_GLOBAL;
        case 2:
                return MOD_PARAM;
        }
        return MOD_LOCAL;
}
```

Nachdem wir nun Befehle im Programmspeicher mit **gen()** ablegen können, sind eine Reihe von weiteren Funktionen zur Code-Generierung sehr leicht zu realisieren, da sie nur **gen()** mit verschiedenen Parametern aufrufen. Diese Funktionen könnten prinzipiell auch als Makros vereinbart werden.

```
gen_alu(mod, comment)
        int mod;                /* ALU Name */
        char * comment;         /* Kommentar */
{
        gen(OP_ALU, mod, 0, comment);
}

gen_li(const)
        char * const;           /* Constant: Wert */
{
        gen(OP_LOAD, MOD_IMMED, atoi(const), const);
}

gen_pr(op, comment)
        int op;                 /* Befehlsname */
        char * comment;         /* Kommentar */
{
        gen(op, 0, 0, comment);
}
```

Constant als Eingabesymbol muß jetzt in einen Zahlenwert umgewandelt werden, bevor der Wert im Programmspeicher abgelegt werden kann. Wir verwenden die Büchereifunktion **atoi()** zur Umwandlung; Konstanten müssen deshalb (wenigstens) der Syntax dieser Funktion genügen. Unser Muster zur lexikalischen Analyse von Konstanten ist in Wirklichkeit noch mehr eingeschränkt, da kein Vorzeichen erlaubt ist.

Vorwärtsverweise bilden das einzige ernsthafte Problem bei Code-Generierung im Speicher. Betrachten wir die typische Situation:

```
if_prefix
        : IF '(' expression rp
                { $$ = gen_jump(OP_JUMPZ, new_label(), "IF"); }

statement
        : if_prefix statement
                { gen_label($1); }
```

new_label() markiert das Ziel für den nötigen Vorwärtssprung; der Funktionswert von **new_label()** wird an **gen_jump()** übergeben. Ein relativ beliebiger Resultatwert von **gen_jump()** wird über den *yacc*-Stack an **gen_label()** weitergereicht. Diese Funktion muß schließlich alles in Ordnung bringen, das heißt, sie muß dafür sorgen, daß der Vorwärtssprung, den **gen_jump()** konstruiert hat, den bei **gen_label()** tatsächlich vorliegenden Wert des Programmzählers **pc** zum Ziel hat.

Folgende Lösung bietet sich an: Wenn **gen_jump()** einen Vorwärtssprung erzeugen soll, kann die Funktion einen Sprung mit beliebigem Sprungziel anlegen. Als Resultat liefert **gen_jump()** dann die Adresse dieses Sprungbefehls im Programmspeicher. **gen_label()** schließlich fügt den Wert des Programmzählers **pc**, also die Adresse bei der der nächste Befehl im Programmspeicher angelegt werden wird, in den Sprungbefehl bei der von **gen_jump()** über den *yacc*-Stack gelieferten Adresse ein, und korrigiert damit den Vorwärtsverweis.

Die Technik funktioniert einwandfrei für Marken als Vorwärtsverweise, die nur einmal verwendet werden, wie dies bei Kontrollstrukturen der Fall ist. **break**-Anweisungen und **continue**-Anweisungen bei bestimmten Schleifen verweisen jedoch möglicherweise mehrmals auf die gleiche Marke.

Offensichtlich muß man alle Verweise auf die gleiche undefinierte Marke im allgemeinen Fall miteinander verketten, damit ein einziger Aufruf von **gen_label()** genügt, um alle Verweise zu korrigieren. Da das Offset-Feld bei einem Vorwärtssprung ungenutzt bleibt, bis es von **gen_label()** korrigiert wird, bietet es sich für die Verkettung an. Eine solche Kette kann nie zur Adresse Null reichen, damit kann Null als Abschluß der Kette verwendet werden.

Im Zusammenhang mit **break** und **continue** können wir aus dem Kontext nicht ableiten, ob das zugehörige Sprungziel auf dem **break**- oder **continue**-Stack bereits existiert und damit sofort eingefügt werden kann, oder ob ein Vorwärtsverweis vorliegt, zu dem man verketten muß. Um diese beiden Fälle zu unterscheiden, markieren wir Vorwärtsverweise, indem wir sie mit *negativen* Indizes durch den Programmspeicher verketten.

gen_jump() muß einen Vorwärtssprung verketten, das heißt, die Adresse des Sprungbefehls im Programmspeicher muß als negativer Resultatwert geliefert werden. Es erweist sich später als nützlich, wenn auch die positive, tatsächliche Zieladresse als Resultat geliefert wird:

```
int gen_jump(op, label, comment)
        int op;                 /* Befehlsname */
        int label;              /* Sprungziel */
        char * comment;         /* Kommentar */
{       int pc = gen(op, 0, label, comment);

        if (label <= 0)
                return -pc;     /* neues Ende der Kette */
        else
                return label;   /* bereits definiert */
}
```

new_label() markiert immer das Ende einer Kette von Vorwärtsverweisen, das heißt, diese Funktion muß konstant Null als Resultat liefern:

```
int new_label()
{
        return 0;               /* Anfang der Kette */
}
```

Beim derzeitigen Stand der Entwicklung sollte **gen_label()** als Argument grundsätzlich eine Kette zur Reparatur vorfinden. Wir werden noch sehen, daß bei einem Aufruf im Kontext von **continue gen_label()** *möglicherweise* nichts zu tun hat. Deshalb wird das Argument von **gen_label()** untersucht; nur wenn es negativ ist, muß eine Kette repariert werden:

```
int gen_label(chain)
        int chain;
{       int next;

        while (chain < 0)
        {       chain = - chain;
                next = prog[chain].p_val;
                if (next > 0)
                        break;  /* schon repariert */
                prog[chain].p_val = pc;
                printf("%d:\t(korr.)\t%d\n", chain, pc);
                chain = next;
        }
        return pc;
}
```

gen_label() muß das Sprungziel, also **pc**, als Resultat liefern, damit die **while**-Anweisung korrekt implementiert ist:

```
loop_prefix
        : WHILE '('
                { $<y_lab>$ = gen_label(new_label());
                  push_continue($<y_lab>$);
                }
          expression rp
                { $$ = $<y_lab>3; }

statement
        : loop_prefix
                { $<y_lab>$ = gen_jump(OP_JUMPZ, new_label(),
                                "WHILE");
                  push_break($<y_lab>$);
                }
          statement
                { gen_jump(OP_JUMP, $1, "zurueck zu WHILE");
                  gen_label($<y_lab>2);
                  pop_break();
                  pop_continue();
                }
```

Betrachten wir die Implementierung von **while** um die Übersetzung von **break** und **continue** zu verstehen. Bei **WHILE** wird der Fortsetzungspunkt als neues Sprungziel definiert: **new_label()** markiert, **gen_label()** korrigiert (nichts, in diesem Fall) und liefert die vereinbarte Adresse als Resultat, die dann auf dem **continue**-Stack deponiert wird:

```
push_continue(label)
        int label;
{
        c_top = push(c_top, label);
}
```

Verweist jetzt eine **continue**-Anweisung auf diese Stack-Spitze, so existiert dort ein positives, definiertes Sprungziel, und die Anweisung wird in einen (Rückwärts-) Sprung übersetzt.

Nach dem **loop_prefix** in **statement** wird ein Vorwärtssprung zum Abbruchpunkt der Schleife generiert: **new_label()** markiert das Ende der zugehörigen Kette, **gen_jump()** erzeugt den Sprung, verkettet ihn und liefert (negativ) seine Adresse im Speicher. Diese Adresse wird auf dem *yacc*-Stack zur späteren Korrektur und auf dem **break**-Stack für **break**-Anweisungen abgelegt:

```
push_break(label)
        int label;
{
        b_top = push(b_top, label);
}
```

Zur Übersetzung von **break**-Anweisungen wird die Spitze im **break**-Stack konsultiert; dort befindet sich jetzt ein undefiniertes Sprungziel. **break** wird folglich in einen Vorwärtssprung übersetzt, der entsprechend verkettet wird.

Im Anschluß an die komplette **while**-Anweisung wird ein **jump** zum (definierten) Fortsetzungspunkt **$1** generiert. Anschließend wird der Abbruchpunkt **$2** durch einen Aufruf von **gen_label()** definiert: da früher ein **jumpz** zu diesem Punkt generiert wurde, wird dieser Vorwärtssprung jetzt korrigiert.

Jetzt können wir den **break**-Stack bearbeiten. Sprünge, die aus **break**-Anweisungen resultieren, können korrigiert werden, wenn sie von **gen_break()** mit dem Stack verkettet wurden:

```
gen_break()
{
        *top(&b_top) = gen_jump(OP_JUMP, *top(&b_top),
                                "BREAK");
}
```

Die Kette beginnt in der Stack-Spitze und führt durch die zu korrigierenden Sprungbefehle. Ein neuer Vorwärtssprung muß entsprechend eingehängt werden: seine Adresse wird von **gen_jump()** geliefert, und muß das Element an der Stack-Spitze ersetzen. Da **gen_jump()** auch die Adresse eines (definierten) Rückwärtssprungs liefert(!), wird das Element an der Stack-Spitze nicht verändert, wenn es auf ein bereits existentes Sprungziel verweist. Wir können folglich **gen_continue()** genau wie **gen_break()** realisieren; beide Anweisungen sind damit korrekt implementiert, unabhängig davon, ob sie in Vor- oder Rückwärtssprünge übersetzt werden:

```
gen_continue()
{
        *top(&c_top) = gen_jump(OP_JUMP, *top(&c_top),
                                "CONTINUE");
}
```

Ein kleines Problem verbleibt jedoch: wir können an das oberste Element auf dem Stack nicht zuweisen, wenn der Stack leer ist. Während die anderen Funktionen zum Stack-Management vom Code-Generator in Kapitel 7 übernommen werden können, muß **top()** Zugriff zur Stack-Beschreibung erhalten, damit ein blindes Element auf einen leeren Stack gebracht werden kann:

```
static int * top(stack)
        struct bc_stack ** stack;
{
        if (! *stack)
        {       error("keine Schleife offen");
                *stack = push(*stack, 0);
        }
        return & (*stack) -> bc_label;
}
```

top() liefert einen *Zeiger* auf den Sprungzieleintrag bei der Stack-Spitze; damit kann dieser Eintrag ohne Kenntnis der Stack-Implementierung verändert werden.

Zurück zum obersten Element im **break**-Stack: wir müssen Vorwärtssprünge zum Abbruchpunkt der Schleife korrigieren, das heißt, **pop_break()** muß **gen_label()** aufrufen:

```
pop_break()
{
        gen_label(*top(&b_top));
        b_top = pop(b_top);
}
```

gen_label() muß hier Vorwärtssprünge korrigieren. **pop_break()** kann am Abbruchpunkt nur aufgerufen werden, wenn solche Vorwärtssprünge wirklich existieren.

Die Kette solcher Sprünge zum Abbruchpunkt führt jedoch zu dem einen **jumpz**-Maschinenbefehl, der im Anschluß an den **loop_prefix** generiert wurde, und der bereits korrigiert wurde! Dies erklärt, warum wir in **gen_label()** die Korrekturschleife bei

```
if (next > 0)
```

abbrechen müssen, das heißt, wenn wir auf der Kette einen Sprungbefehl erreichen, der bereits korrigiert wurde.

continue funktioniert analog zu **break**. Der **continue**-Stack wird daher genauso bearbeitet, obgleich bei unserer Anwendung **continue** keine Vorwärtssprünge erzeugen kann:

```
pop_continue()
{
        gen_label(*top(&c_top));
        c_top = pop(c_top);
}
```

Durch Funktionsaufrufe können sich ebenfalls Vorwärtsverweise ergeben: in C dürfen Funktionen aufgerufen werden, bevor sie definiert sind. In solchen Fällen ist ein **call**-Maschinenbefehl ein Vorwärtsverweis. Ist die Funktion einmal definiert, müssen wir uns die Adresse ihres **entry**-Befehls merken; spätere Aufrufe der Funktionen können dann sofort richtig codiert werden.

Eine beliebte Technik zur Lösung dieses Problems ist ein *Transfervektor*: Alle Funktionsaufrufe erfolgen indirekt durch einen Vektor von Funktionsadressen, der sich in einer global bekannten Position im Speicher befindet, und dessen Elemente im Zuge der Definition der Funktionen jeweils initialisiert werden.

Unsere Technik ist effizienter: wir verketten alle Vorwärtsverweise durch **call** und verwenden **gen_label()** zur Korrektur, wenn der entsprechende **entry**-Ma-

schinenbefehl generiert wird. Der Anfang der Kette sowie die Adresse des **entry**-Befehls, wenn diese erst einmal bekannt ist, kann in der **s_offset**-Komponente im Symboltabelleneintrag der Funktion abgelegt werden. Diese Komponente wird mit Null initialisiert und ist daher ein akzeptabler Abschluß der Kette.

```
gen_call(symbol, count)
        struct symtab * symbol; /* Funktion */
        int     count;          /* Anzahl Argumente */
{       int pc;

        chk_parm(symbol, count);
        pc = gen(OP_CALL, count, symbol->s_offset,
                symbol->s_name);
        if (symbol->s_offset <= 0)
                symbol->s_offset = -pc; /* Ende der Kette */
        while (count-- > 0)
                gen_pr(OP_POP, "Argument entfernen");
        gen(OP_LOAD, MOD_GLOBAL, 0, "Resultat auf Stack");
}
```

gen_entry() muß etwaige Vorwärtsverweise mit Hilfe von **gen_label()** korrigieren und dann die Anfangsadresse der Funktion in **s_offset** eintragen. Der **entry**-Maschinenbefehl verursacht ein neues Problem: hier benötigen wir die Größe des lokalen Bereichs, die am Anfang der Funktion, bei der Generierung von **entry**, noch unbekannt ist. Wieder einmal liefern wir als Funktionsresultat die Adresse eines später zu korrigierenden Maschinenbefehls, legen diese Adresse auf dem *yacc*-Stack ab und korrigieren schließlich in der **fix_entry()**-Funktion, die im folgenden definiert wird.

```
int gen_entry(symbol)
        struct symtab * symbol; /* Funktion */
{
        symbol->s_offset = gen_label(symbol->s_offset);
        gen(OP_ENTRY, 0, 0, symbol->s_name);
        return symbol->s_offset;
}

fix_entry(symbol, label)
        struct symtab * symbol; /* Funktion */
        int label;
{       extern int l_max;    /* Groesse des lokalen Bereichs */

        prog[label].p_val = l_max;
        printf("%d:\tentry\t%d\t; %s\n",
                label, l_max, symbol->s_name);
}
```

Damit ist die Code-Generierung vollständig. Wenn wir **end_program()** erreichen, kennen wir die Größe des generierten Programms und die Größe des notwendigen globalen Bereichs, und wir können die Startadresse der **main()**-Funktion der Symboltabelle entnehmen. Mit dieser Information können wir dann den Simulator aufrufen:

```
end_program()
{       extern int g_offset; /* Groesse des globalen Bereichs */
        extern struct symtab * s_find();
        int main = s_find("main") -> s_offset;

        all_program();          /* globale Variablen anlegen */
        printf("%d:\tend\t%d,%d\n", pc, g_offset, main);
        simulate(pc, g_offset, main);
}
```

s_find() kann immer einen Symboltabelleneintrag für **main()** finden; falls jedoch die Funktion nicht definiert wurde, enthält **s_offset** Null. Die Simulation beginnt dann bei Adresse **0**, das heißt, der Simulator wird sofort angehalten.[1]

Ob wir den Simulator starten, wenn bei der Übersetzung Fehler entdeckt wurden, kann immer noch entschieden werden: **yynerrs** enthält die Zahl der entdeckten Fehler; diese **int**-Variable wird auch von unserer eigenen **error()**-Funktion verändert. Wir plädieren dafür, den Simulator in jedem Fall zu starten - wir haben versucht ihn robust genug zu gestalten, daß er auch bei groben Fehlern zur Laufzeit wenigstens mit einer entsprechenden Meldung abgebrochen wird.

8.3 Beispiel

Als Beispiel betrachten wir folgende rekursive Formulierung des Euklid'schen Algorithmus':

```
/*
 *      Euklid's Algorithmus (rekursiv)
 */

main()
{
        gcd(36,54);
}
```

[1] Falls **main()** undefiniert ist, aber aufgerufen wurde, ist **s_offset** negativ. Dieser Fall führt im Simulator nicht ganz korrekt zu einem Aufruf von **bug()**. A.d.Ü.

```
gcd(a,b)
{
        if (a == b)
                return a;
        else if (a > b)
                return gcd(a-b, b);
        else
                return gcd(a, b-a);
}
```

Übersetzung und Ausführung dieses Programms führt zu folgendem Protokoll:

```
1:      10      0,0     ; main
2:      4       4,36    ; 36
3:      4       4,54    ; 54
4:      9       2,0     ; gcd
5:      6       0,0     ; Argument entfernen
6:      6       0,0     ; Argument entfernen
7:      4       1,0     ; Resultat auf Stack
8:      6       0,0     ; Stack abraeumen
9:      11      0,0     ; Funktionsende
1:      entry   0       ; main
4:      (korr.) 10
10:     10      0,0     ; gcd
11:     4       2,0     ; a
12:     4       2,1     ; b
13:     1       10,0    ; ==
14:     7       0,0     ; IF
15:     4       2,0     ; a
16:     5       1,0     ; Resultat speichern
17:     11      0,0     ; RETURN
18:     8       0,0     ; ueber ELSE hinweg
14:     (korr.) 19
19:     4       2,0     ; a
20:     4       2,1     ; b
21:     1       7,0     ; >
22:     7       0,0     ; IF
23:     4       2,0     ; a
24:     4       2,1     ; b
25:     1       2,0     ; -
26:     4       2,1     ; b
27:     9       2,10    ; gcd
28:     6       0,0     ; Argument entfernen
29:     6       0,0     ; Argument entfernen
30:     4       1,0     ; Resultat auf Stack
31:     5       1,0     ; Resultat speichern
```

```
32:     11      0,0     ; RETURN
33:     8       0,0     ; ueber ELSE hinweg
22:     (korr.) 34
34:     4       2,0     ; a
35:     4       2,1     ; b
36:     4       2,0     ; a
37:     1       2,0     ; -
38:     9       2,10    ; gcd
39:     6       0,0     ; Argument entfernen
40:     6       0,0     ; Argument entfernen
41:     4       1,0     ; Resultat auf Stack
42:     5       1,0     ; Resultat speichern
43:     11      0,0     ; RETURN
33:     (korr.) 44
18:     (korr.) 44
44:     11      0,0     ; Funktionsende
10:     entry   0       ; gcd
45:     end     1,1
Ausfuehrung beginnt...
4:      call    10
14:     jumpz   19
22:     jumpz   34
38:     call    10
14:     jumpz   19
27:     call    10
16:     store   1,0     to 18
17:     return  18 to 28
31:     store   1,0     to 18
32:     return  18 to 39
42:     store   1,0     to 18
43:     return  18 to 5
9:      return  18 to 0
0:      halt
```

Das Beispiel zeigt sehr gut, daß der Code durch einen Postprozessor wesentlich verkürzt werden könnte. Von einer Serie von **return**-Befehlen zum Beispiel kann nur der erste wirklich erreicht werden.

8.4 Aufgaben

1. Verbessern Sie den Simulator dadurch, daß **return** Resultatwerte auf dem Stack übergibt. In **gen_call()** brauchen dann keine **pop**- und **load**-Befehle im Anschluß an **call** generiert zu werden. (Vergleiche Abschnitt 8.1.)

2. Wie erwähnt, sind die **call**-, **return**-, und **entry**-Befehle unserer *kleinC*-Maschine nicht sehr realistisch. Bei wirklichen Maschinen sind diese Befehle wesentlich primi-

tiver. Entwerfen Sie 'realistische' Maschinenbefehle und ändern Sie den Code-Generator und Simulator entsprechend. Hinweis: Sie werden wahrscheinlich Befehle zur Manipulation von Registern benötigen. ✏

3. Fügen Sie Optionen zur Steuerung des Simulators hinzu, entweder als Optionen beim Aufruf des Systems, oder durch spezielle Kommentare, die zur Generierung entsprechender Pseudobefehle führen. Optionen könnten zum Beispiel folgendes beeinflussen: Protokoll der generierten Befehle vor Beginn der Simulation; Ausführung des Programms, falls bei der Übersetzung Fehler festgestellt werden; Ausführung in Abhängigkeit von Zahl und Art dieser Fehler; Begrenzung der Anzahl der Simulationszyklen; Ablaufverfolgung im Protokoll, zum Beispiel durch Wahl von Art und Anzahl der verfolgten Maschinenbefehle. ✏

4. Ändern Sie den Simulator so, daß arithmetische Fehler wie z.B. Division durch Null behandelt werden. ✏

5. Ändern Sie den Simulator so, daß Speicher dynamisch beschafft wird. Damit entfallen die Vektoren mit 'änderbarer' Größe, die am Anfang des Kapitels definiert wurden. ✏

6. Die Struktur eines Elements im Programmspeicher, die am Anfang des Kapitels zur Vereinfachung der Decodierung vereinbart wurde, ist nicht gerade repräsentativ für normale Maschinenarchitekturen, und sie ist auch nicht besonders kompakt. Ändern Sie diesen Aspekt des Simulators, damit das System effizienter und realistischer wird. ✏

8.5 Projekte

1. Reorganisieren Sie den Compiler so, daß er aus mehreren einzelnen Programmen besteht: (a) dem Compiler aus Kapitel 7, der Assembler-Text generiert; (b) einem einfachen Assembler (der natürlich mit Hilfe von *yacc* und *lex* entwickelt wird!); und (c) dem Simulator. ✏

2. Mit den drei Programmen aus dem vorhergehenden Projekt kann der Compiler so umgebaut werden, daß Funktionen separat übersetzt und dann entweder mit dem Assembler oder mit einem zusätzlichen Lader zusammengefügt werden können. Beachten Sie dabei, daß globale Variablen bei jeder Übersetzung vorkommen können, und dann insgesamt im endgültigen, zusammengesetzten Programm vorhanden sein müssen. ✏

3. Viele Spracherweiterungen sind natürlich denkbar: **char** oder andere Integer-Variablen, Gleitkomma-Variablen, Vektoren mit oder ohne Zeiger, Strukturen und schließlich andere Kontrollstrukturen wie **switch**, **for** oder **do while**. ✏

4. Weitere interessante Spracherweiterungen ergeben sich, wenn man parallele Verarbeitung einführt, zum Beispiel mit Hilfe von Standardfunktionen (das heißt, neuen Maschinenbefehlen im Simulator) für Coroutine-Sprünge im Stil von Modula-2, oder mit einer **parallel**-Kontrollstruktur. Bei diesem Projekt sollte man auch Vektoren einführen, um das Resultat realistisch zu gestalten. ✏

Anhang A: "kleinC"-Compiler

Die einzelnen Kapitel des Buches dokumentieren meistens nur Änderungen am Compiler. Dieser Anhang enthält die kompletten Quellen, damit sie im Zusammenhang betrachtet werden können.

A.1 Syntaxanalyse

yacc konstruiert den Parser aus der Datei *kleinc.y*. Diese Datei muß erweitert werden während die verschiedenen Phasen des Compilers, nämlich Erkennung, Fehlerbehandlung, Semantikanalyse und Synthese, entwickelt werden. Hier folgt nun die endgültige Fassung der Datei für Code-Generierung. Für Simulation ergibt sich nur ein kleiner Unterschied: die Preprozessor-Anweisung

```
#include "gen.h"
```

wird ersetzt durch

```
#include "sim.h"
```

Die zwei Definitionsdateien vereinbaren identische Namen, aber mit recht unterschiedlichen Definitionen.

kleinc.y

```
/*
 *      kleinC
 *      Syntaxanalyse und Fehlerbehandlung
 *      Symboltabelle
 *      Speichermanagement
 *      Code-Generierung
 *      (s/r Konflikte: bei ELSE, bei error)
 */

%{
#include "symtab.h"     /* Symboltabelle */
#include "gen.h"        /* Code-Generierung */

#define OFFSET(x)       ( ((struct symtab *) x) -> s_offset )
#define NAME(x)         ( ((struct symtab *) x) -> s_name )

extern int l_offset, l_max;
%}
%union  {
        struct symtab * y_sym;  /* Identifier */
        char * y_str;           /* Constant */
        int y_num;              /* Zaehler */
        int y_lab;              /* Marke */
        }
```

```
/*
 *      Eingabesymbole
 */

%token  <y_sym> Identifier
%token  <y_str> Constant
%token  INT
%token  IF
%token  ELSE
%token  WHILE
%token  BREAK
%token  CONTINUE
%token  RETURN
%token  ';'
%token  '('
%token  ')'
%token  '{'
%token  '}'
%token  '+'
%token  '-'
%token  '*'
%token  '/'
%token  '%'
%token  '>'
%token  '<'
%token  GE      /* >= */
%token  LE      /* <= */
%token  EQ      /* == */
%token  NE      /* != */
%token  '&'
%token  '^'
%token  '|'
%token  '='
%token  PE      /* += */
%token  ME      /* -= */
%token  TE      /* *= */
%token  DE      /* /= */
%token  RE      /* %= */
%token  PP      /* ++ */
%token  MM      /* -- */
%token  ','
```

```
/*
 *      Syntaxbegriffe mit Typ
 */

%type   <y_sym> optional_parameter_list, parameter_list
%type   <y_num> optional_argument_list, argument_list
%type   <y_lab> if_prefix, loop_prefix

/*
 *      Vorrangtabelle
 */

%right  '=' PE ME TE DE RE
%left   '|'
%left   '^'
%left   '&'
%left   EQ NE
%left   '<' '>' GE LE
%left   '+' '-'
%left   '*' '/' '%'
%right  PP MM

%%

program
        :       { init(); }
          definitions
                { end_program(); }

definitions
        : definition
        | definitions definition
                { yyerrok; }
        | error
        | definitions error

definition
        : function_definition
        | INT function_definition
        | declaration
```

```
function_definition
        : Identifier '('
                { make_func($1);
                  blk_push();
                }
          optional_parameter_list rp
          parameter_declarations
                { chk_parm($1, parm_default($4));
                  all_parm($4);
                  l_max = 0;
                  $<y_lab>$ = gen_entry($1);
                }
          compound_statement
                { all_func($1);
                  gen_pr(OP_RETURN, "Funktionsende");
                  fix_entry($1, $<y_lab>7);
                }

optional_parameter_list
        : /* keine Parameter */
                { $$ = (struct symtab *) 0; }
        | parameter_list
                /* $$ = $1 = Kette der Parameter */

parameter_list
        : Identifier
                { $$ = link_parm($1, (struct symtab *) 0); }
        | Identifier ',' parameter_list
                { $$ = link_parm($1, $3);
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | error parameter_list
                { $$ = $2; }
        | Identifier error parameter_list
                { $$ = link_parm($1, $3); }
        | error ',' parameter_list
                { $$ = $3;
                  yyerrok;
                }
```

```
parameter_declarations
        : /* null */
        | parameter_declarations parameter_declaration
                { yyerrok; }
        | parameter_declarations error

parameter_declaration
        : INT parameter_declarator_list sc

parameter_declarator_list
        : Identifier
                { make_parm($1); }
        | parameter_declarator_list ',' Identifier
                { make_parm($3);
                  yyerrok;
                }
        | error
        | parameter_declarator_list error
        | parameter_declarator_list error Identifier
                { make_parm($3);
                  yyerrok;
                }
        | parameter_declarator_list ',' error

compound_statement
        : '{'
                { $<y_lab>$ = l_offset;
                  blk_push();
                }
          declarations statements rr
                { if (l_offset > l_max)
                        l_max = l_offset;
                  l_offset = $<y_lab>2;
                  blk_pop();
                }

declarations
        : /* null */
        | declarations declaration
                { yyerrok; }
        | declarations error

declaration
        : INT declarator_list sc
```

```
declarator_list
        : Identifier
                { all_var($1); }
        | declarator_list ',' Identifier
                { all_var($3);
                  yyerrok;
                }
        | error
        | declarator_list error
        | declarator_list error Identifier
                { all_var($3);
                  yyerrok;
                }
        | declarator_list ',' error

statements
        : /* null */
        | statements statement
                { yyerrok; }
        | statements error

statement
        : expression sc
                { gen_pr(OP_POP, "Stack abraeumen"); }
        | sc
        | BREAK sc
                { gen_break(); }
        | CONTINUE sc
                { gen_continue(); }
        | RETURN sc
                { gen_pr(OP_RETURN, "RETURN"); }
        | RETURN expression sc
                { gen(OP_STORE, MOD_GLOBAL, 0,
                        "Resultat speichern");
                  gen_pr(OP_RETURN, "RETURN");
                }
        | compound_statement
        | if_prefix statement
                { gen_label($1); }
        | if_prefix statement ELSE
                { $<y_lab>$ = gen_jump(OP_JUMP, new_label(),
                        "ueber ELSE hinweg");
                  gen_label($1);
                }
          statement
                { gen_label($<y_lab>4); }
```

```
        | loop_prefix
                { $<y_lab>$ = gen_jump(OP_JUMPZ, new_label(),
                        "WHILE");
                  push_break($<y_lab>$);
                }
          statement
                { gen_jump(OP_JUMP, $1, "zurueck zu WHILE");
                  gen_label($<y_lab>2);
                  pop_break();
                  pop_continue();
                }

if_prefix
        : IF '(' expression rp
                { $$ = gen_jump(OP_JUMPZ, new_label(), "IF"); }
        | IF error
                { $$ = gen_jump(OP_JUMPZ, new_label(), "IF"); }

loop_prefix
        : WHILE '('
                { $<y_lab>$ = gen_label(new_label());
                  push_continue($<y_lab>$);
                }
          expression rp
                { $$ = $<y_lab>3; }
        | WHILE error
                { $$ = gen_label(new_label());
                  push_continue($$);
                }

expression
        : binary
        | expression ','
                { gen_pr(OP_POP, "abraeumen"); }
          binary
                { yyerrok; }
        | error ',' binary
                { yyerrok; }
        | expression error
        | expression ',' error
```

```
binary
        : Identifier
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
        | Constant
                { gen_li($1); }
        | '(' expression rp
        | '(' error rp
        | Identifier '('
                { chk_func($1); }
          optional_argument_list rp
                { gen_call($1,$4); }
        | PP Identifier
                { chk_var($2);
                  gen(OP_INC, gen_mod($2), OFFSET($2),
                        NAME($2));
                }
        | MM Identifier
                { chk_var($2);
                  gen(OP_DEC, gen_mod($2), OFFSET($2),
                        NAME($2));
                }
        | binary '+' binary
                { gen_alu(ALU_ADD, "+"); }
        | binary '-' binary
                { gen_alu(ALU_SUB, "-"); }
        | binary '*' binary
                { gen_alu(ALU_MUL, "*"); }
        | binary '/' binary
                { gen_alu(ALU_DIV, "/"); }
        | binary '%' binary
                { gen_alu(ALU_MOD, "%"); }
        | binary '>' binary
                { gen_alu(ALU_GT, ">"); }
        | binary '<' binary
                { gen_alu(ALU_LT, "<"); }
        | binary GE binary
                { gen_alu(ALU_GE, ">="); }
        | binary LE binary
                { gen_alu(ALU_LE, "<="); }
        | binary EQ binary
                { gen_alu(ALU_EQ, "=="); }
```

```
        | binary NE binary
                { gen_alu(ALU_NE, "!="); }
        | binary '&' binary
                { gen_alu(ALU_AND, "&"); }
        | binary '^' binary
                { gen_alu(ALU_XOR, "^"); }
        | binary '|' binary
                { gen_alu(ALU_OR, "|"); }
        | Identifier '=' binary
                { chk_var($1);
                  gen(OP_STORE, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
        | Identifier PE
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
          binary
                { gen_alu(ALU_ADD, "+");
                  gen(OP_STORE, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
        | Identifier ME
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
          binary
                { gen_alu(ALU_SUB, "-");
                  gen(OP_STORE, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
        | Identifier TE
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
          binary
                { gen_alu(ALU_MUL, "*");
                  gen(OP_STORE, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
```

```
        | Identifier DE
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
          binary
                { gen_alu(ALU_DIV, "/");
                  gen(OP_STORE, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
        | Identifier RE
                { chk_var($1);
                  gen(OP_LOAD, gen_mod($1), OFFSET($1),
                        NAME($1));
                }
          binary
                { gen_alu(ALU_MOD, "%");
                  gen(OP_STORE, gen_mod($1), OFFSET($1),
                        NAME($1));
                }

optional_argument_list
        : /* keine Argumente */
                { $$ = 0; }
        | argument_list
                /* $$ = $1 = Anzahl Argumente */

argument_list
        : binary
                { $$ = 1; }
        | argument_list ',' binary
                { ++ $$;
                  yyerrok;
                }
        | error
                { $$ = 0; }
        | argument_list error
        | argument_list ',' error
```

```
/*
 *      bestimmte Eingabesymbole nie ignorieren
 */

rp      : ')'   { yyerrok; }
sc      : ';'   { yyerrok; }
rr      : '}'   { yyerrok; }
```

Der Parser wird dann mit folgendem Kommando konstruiert:

```
yacc -d kleinc.y
```

-d instruiert *yacc* sowohl den Parser *y.tab.c* als auch die Definitionsdatei *y.tab.h* zu produzieren; letztere enthält die Definitionen, die sich aus den **%token**-Anweisungen ergeben sowie die Typdefinition für den Wert-Stack.

A.2 Lexikalische Analyse

Die Datei *kleinc.l* wurde im Abschnitt 2.7 vorgestellt. Aus dieser Datei wird die Funktion **yylex()** zur lexikalischen Analyse mit folgendem Kommando konstruiert:

```
lex kleinc.l
```

A.3 Meldungen

message.c enthält die C-Funktionen zur Ausgabe von Meldungen, die im Kapitel 5 vorgeführt wurden. (Kommentare wie /***VARARGS1***/ sorgen dafür, daß *lint* nicht bemängelt, daß diese Funktionen im Stil von **printf()** mit einer wechselnden Anzahl von Argumenten aufgerufen werden.)

message.c

```
/*
 *      Meldungen
 */

#include <stdio.h>

#define VARARG  fmt, v1, v2, v3, v4, v5
#define VARPARM (VARARG) char * fmt;

extern FILE * yyerfp;
```

```
                                        /*VARARGS1*/
message VARPARM
{
        yywhere();
        fprintf(yyerfp, VARARG);
        putc('\n', yyerfp);
}

                                        /*VARARGS1*/
error VARPARM
{       extern int yynerrs;

        fprintf(yyerfp, "[Fehler %d] ", ++ yynerrs);
        message(VARARG);
}

                                        /*VARARGS1*/
warning VARPARM
{
        fputs("[Warnung] ", yyerfp);
        message(VARARG);
}

                                        /*VARARGS1*/
fatal VARPARM
{
        fputs("[Abbruch] ", yyerfp);
        message(VARARG);
        exit(1);
}

                                        /*VARARGS1*/
bug VARPARM
{
        fputs("SYSTEMFEHLER: ", yyerfp);
        message(VARARG);
        exit(1);
}
```

In der Datei *message.c* kann auch die **strsave()**-Funktion definiert werden, falls sie nicht bereits in der lokalen C-Bücherei verfügbar ist. **strsave()** könnte auch in der Bücherei bereitgestellt werden, die im Abschnitt A.8 diskutiert wird. Hier ist eine mögliche Definition für **strsave()**:

```
char * strsave(s)
        register char * s;
{       register char * cp = calloc(strlen(s)+1, 1);

        if (cp)
                return strcpy(cp, s);
        fatal("Kein Platz mehr fuer Strings.");
}
```

A.4 Symboltabelle

Die Datei *symtab.h* enthält die Vereinbarungen für die Symboltabelle, die zur Übersetzung von *kleinc.y* notwendig sind.

symtab.h

```
/*
 *      kleinC -- Definitionsdatei zur Symboltabelle
 */

struct symtab {
        char *  s_name;         /* -> Name */
        int     s_type;         /* Symboltyp */
        int     s_blknum;       /* statisch verschachtelt */
        union {                 /* s.u. */
                int s__num;
                struct symtab * s__link;
                } s__;
        int     s_offset;       /* Symboldefinition */
        struct symtab * s_next; /* -> naechstes Symbol */
        };

#define s_pnum  s__.s__num      /* Anzahl Parameter */
#define NOT_SET (-1)            /* Anzahl unbekannt */
#define s_plist s__.s__link     /* Kette der Parameter */

/*
 *      Werte fuer s_type
 */

#define UDEC    0       /* nicht vereinbart */
#define FUNC    1       /* Funktion */
#define UFUNC   2       /* undefinierte Funktion */
#define VAR     3       /* definierte Variable */
#define PARM    4       /* undeklarierter Parameter */
```

```
/*
 *      s_type Darstellung fuer S_TRACE
 */

#define SYMmap  "nicht vereinbart", "Funktion", \
                "undefinierte Funktion", "Variable", \
                "Parameter"

/*
 *      Typvereinbarungen fuer Funktionen der Symboltabelle
 */

struct symtab * link_parm();  /* Parameter verketten */
struct symtab * s_find();     /* Symbol nach Namen aufsuchen */
struct symtab * make_parm();  /* Parameter deklarieren */
struct symtab * make_var();   /* Variable definieren */
struct symtab * make_func();  /* Funktion definieren */

/*
 *      Typvereinbarungen fuer Buechereifunktionen
 */

char * strsave();       /* Zeichenkette dynamisch speichern */
char * calloc();        /* dynamisch Speicher beschaffen */
```

Die Datei *symtab.c* enthält die C-Funktionen zur Manipulation der Symboltabelle, die in Kapitel 5 beschrieben wurden.

symtab.c

```
/*
 *      kleinC -- Symboltabellen-Manager
 */

#include "symtab.h"
#include "y.tab.h"

/*
 *      Symboltabelle
 */

static  struct  symtab
        symtab,                 /* blindes Element */
        * s_gbl;                /* Ende der globalen Kette */
#define s_lcl   (& symtab)      /* Ende der lokalen Kette */
```

```
/*
 *      Blocktabelle
 */

static int blknum = 0;          /* Verschachtelungstiefe */

/*
 *      neuer Name im lokalen Bereich
 */

static struct symtab * s_create(name)
        register char * name;
{       register struct symtab * new_entry = (struct symtab *)
                        calloc(1, sizeof(struct symtab));

        if (new_entry)
        {       new_entry->s_next = s_lcl->s_next;
                s_lcl->s_next = new_entry;
                new_entry->s_name = strsave(name);
                new_entry->s_type = UDEC;
                new_entry->s_blknum = 0;
                new_entry->s_pnum = NOT_SET;
                return new_entry;
        }
        fatal("Kein Platz mehr fuer Symbole.");
        /*NOTREACHED*/
}

/*
 *      lokalen Eintrag in den globalen Bereich verlagern
 */

static s_move(symbol)
        register struct symtab * symbol;
{       register struct symtab * ptr;

        /* auf der Kette suchen (muss vorhanden sein!) */
        for (ptr=s_lcl; ptr->s_next != symbol; ptr=ptr->s_next)
                if (! ptr->s_next)
                        bug("s_move");

        /* ausketten */
        ptr->s_next = symbol->s_next;
```

```
        /* am globalen Ende einhaengen */
        s_gbl->s_next = symbol;
        s_gbl = symbol;
        s_gbl->s_next = (struct symtab *) 0;
}

/*
 *      Symbol- und Block-Tabelle initialisieren
 */

init()
{
        blk_push();
        s_gbl = s_create("main");
        s_gbl->s_type = UFUNC;
}

/*
 *      neuer Block
 */

blk_push()
{
        ++ blknum;
}

/*
 *      Eintrag nach Namen finden
 */

struct symtab * s_find(name)
        char * name;
{       register struct symtab * ptr;

        /* Symboltabelle absuchen */
        for (ptr = s_lcl->s_next; ptr; ptr = ptr->s_next)
                if (! ptr->s_name)
                        bug("s_find");
                else
                        /* bei gleichen Namen: -> Eintrag */
                        if (strcmp(ptr->s_name, name) == 0)
                                return ptr;
        /* Suche erfolglos: NULL */
        return (struct symtab *) 0;
}
```

```
/*
 *      Schnittstelle zur lexikalischen Analyse:
 *              Identifier finden oder eintragen,
 *              Text einer Constant speichern
 */

s_lookup(yylex)
        int yylex;              /* Constant oder Identifier */
{       extern char yytext[];   /* Darstellung des Symbols */

        switch (yylex) {
        case Constant:
                yylval.y_str = strsave(yytext);
                break;
        case Identifier:
                if (yylval.y_sym = s_find(yytext))
                        break;
                yylval.y_sym = s_create(yytext);
                break;
        default:
                bug("s_lookup");
        }
}

/*
 *      Eintrag als Teil einer parameter_list markieren
 */

struct symtab * link_parm(symbol, next)
        register struct symtab * symbol, * next;
{
        switch (symbol->s_type) {
        case PARM:
                error("Parameter %s doppelt", symbol->s_name);
                return next;
        case FUNC:
        case UFUNC:
        case VAR:
                symbol = s_create(symbol->s_name);
        case UDEC:
                break;
        default:
                bug("link_parm");
        }
```

```
        symbol->s_type = PARM;
        symbol->s_blknum = blknum;
        symbol->s_plist = next;
        return symbol;
}

/*
 *      Parameter vereinbaren
 */

struct symtab * make_parm(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case VAR:
                if (symbol->s_blknum == 2)
                {       error("Parameter %s zweimal deklariert",
                                symbol->s_name);
                        return symbol;
                }
        case UDEC:
        case FUNC:
        case UFUNC:
                error("%s ist kein Parameter", symbol->s_name);
                symbol = s_create(symbol->s_name);
        case PARM:
                break;
        default:
                bug("make_parm");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
        return symbol;
}
```

```
/*
 *      Variable definieren
 */

struct symtab * make_var(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case VAR:
        case FUNC:
        case UFUNC:
                if (symbol->s_blknum == blknum
                    || symbol->s_blknum == 2 && blknum == 3)
                        error("Name %s zweimal vereinbart",
                                symbol->s_name);
                symbol = s_create(symbol->s_name);
        case UDEC:
                break;
        case PARM:
                error("Parameter %s nicht vorgesehen",
                        symbol->s_name);
                break;
        default:
                bug("make_var");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
        return symbol;
}

/*
 *      Funktion definieren
 */

struct symtab * make_func(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UFUNC:
        case UDEC:
                break;
```

```
        case VAR:
                error("Funktion %s dupliziert globale Variable",
                        symbol->s_name);
                return symbol;
        case FUNC:
                error("Function %s zweimal definiert",
                        symbol->s_name);
                return symbol;
        default:
                bug("make_func");
        }
        symbol->s_type = FUNC;
        symbol->s_blknum = 1;
        return symbol;
}

/*
 *      Anzahl Parameter definieren oder verifizieren
 */

chk_parm(symbol, count)
        register struct symtab * symbol;
        register int count;
{
        if (symbol->s_pnum == NOT_SET)
                symbol->s_pnum = count;
        else if ((int) symbol->s_pnum != count)
             warning("Funktion %s hat eigentlich %d Argument(e)",
                        symbol->s_name, symbol->s_pnum);
}

/*
 *      Voreinstellung fuer nicht deklarierte Parameter
 *      Parameter zaehlen
 */

int parm_default(symbol)
        register struct symtab * symbol;
{       register int count = 0;

        while (symbol)
        {       ++ count;
                if (symbol->s_type == PARM)
                        symbol->s_type = VAR;
                symbol = symbol->s_plist;
        }
        return count;
}
```

```
/*
 *      Block beenden
 */

blk_pop()
{       register struct symtab * ptr;

        for (ptr = s_lcl->s_next;
             ptr &&
             (ptr->s_blknum >= blknum || ptr->s_blknum == 0);
             ptr = s_lcl->s_next)
        {
                if (! ptr->s_name)
                        bug("blk_pop: Name ist NULL");
#ifdef TRACE
        {       static char * type[] = { SYMmap };

                message(
                    "Entfernt wird %s: %s, Tiefe %d, Offset %d",
                        ptr->s_name, type[ptr->s_type],
                        ptr->s_blknum, ptr->s_offset);
        }
#endif TRACE
                if (ptr->s_type == UFUNC)
                        error("undefinierte Funktion %s",
                                ptr->s_name);
                cfree(ptr->s_name);
                s_lcl->s_next = ptr->s_next;
                cfree(ptr);
        }
        -- blknum;
}

/*
 *      Verweis oder Zuweisung fuer Variable pruefen
 */

chk_var(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UDEC:
                error("undefinierte Variable %s",
                        symbol->s_name);
                break;
```

```
        case PARM:
                error("Parameter %s unerwartet",
                        symbol->s_name);
                break;
        case FUNC:
        case UFUNC:
                error("Funktion %s als Variable verwendet",
                        symbol->s_name);
        case VAR:
                return;
        default:
                bug("check_var");
        }
        symbol->s_type = VAR;
        symbol->s_blknum = blknum;
}

/*
 *      Verweis auf Funktion pruefen, evtl. implizit vereinbaren
 */

chk_func(symbol)
        register struct symtab * symbol;
{
        switch (symbol->s_type) {
        case UDEC:
                break;
        case PARM:
                error("Parameter %s unerwartet",
                        symbol->s_name);
                symbol->s_pnum = NOT_SET;
                return;
        case VAR:
                error("Variable %s als Funktion verwendet",
                        symbol->s_name);
                symbol->s_pnum = NOT_SET;
        case UFUNC:
        case FUNC:
                return;
        default:
                bug("check_func");
        }
        s_move(symbol);
        symbol->s_type = UFUNC;
        symbol->s_blknum = 1;
}
```

A.5 Speichermanagement

Die C-Funktionen zum Speichermanagement aus dem Kapitel 6 befinden sich in der Datei *mem.c*.

mem.c

```
/*
 *      kleinC -- Speichermanagement
 */

#include "symtab.h"

/*
 *      globale Zaehler
 */

int     g_offset = 1,    /* Offset im globalen Bereich */
        l_offset = 0,    /* Offset im lokalen Bereich */
        l_max;           /* Groesse des lokalen Bereichs */

/*
 *      (globale oder lokale) Variable bereitstellen
 */

all_var(symbol)
        register struct symtab * symbol;
{       extern struct symtab * make_var();

        symbol = make_var(symbol);

        /* falls kein Parameter:
           im entsprechenden Bereich anlegen */
        switch (symbol->s_blknum) {
        default:                        /* lokal */
                symbol->s_offset = l_offset++;
        case 2:                         /* Parameter */
                break;
        case 1:                         /* global */
                symbol->s_offset = g_offset++;
                break;
        case 0:
                bug("all_var");
        }
}
```

```
/*
 *      gesamte Speicherdefinition abschliessen
 */

all_program()
{
        blk_pop();

#ifdef  TRACE
        message("globaler Bereich umfasst %d Wort(e)",
                g_offset);
#endif
}

/*
 *      alle Parameter bereitstellen
 */

all_parm(symbol)
        register struct symtab * symbol;
{       register int p_offset = 0;

        while (symbol)
        {       symbol->s_offset = p_offset ++;
                symbol = symbol->s_plist;
        }

#ifdef  TRACE
        message("Parameterbereich umfasst %d Wort(e)",
                p_offset);
#endif
}

/*
 *      Speicherdefinition fuer Funktion abschliessen
 */

all_func(symbol)
        struct symtab * symbol;
{
        blk_pop();

#ifdef  TRACE
        message("lokaler Bereich umfasst %d Wort(e)", l_max);
#endif
}
```

A.6 Code-Generierung

Die Definitionsdatei *gen.h* enthält die Namen der Maschinenbefehle, die zur symbolischen Code-Generierung benötigt werden.

gen.h

```
/*
 *      kleinC -- Definitionsdatei zur Code-Generierung
 */

/*
 *      Befehlsnamen fuer fiktive Maschine
 */

                    /* Befehl */  /* Operanden              */
#define OP_ALU      "alu"         /* Arithmetik-Logik-Op    */
#define OP_DEC      "dec"         /* Bereich,Offset         */
#define OP_INC      "inc"         /* Bereich,Offset         */
#define OP_LOAD     "load"        /* Bereich,Offset         */
#define OP_STORE    "store"       /* Bereich,Offset         */
#define OP_POP      "pop"         /*                        */
#define OP_JUMPZ    "jumpz"       /* Marke                  */
#define OP_JUMP     "jump"        /* Marke                  */
#define OP_CALL     "call"        /* Anz-Argum,Adresse      */
#define OP_ENTRY    "entry"       /* lokaler-Bereich        */
#define OP_RETURN   "return"      /*                        */

/*
 *      Bereichsnamen
 */

#define MOD_GLOBAL  "gbl"         /* globale Variable       */
#define MOD_PARAM   "par"         /* Parameter              */
#define MOD_LOCAL   "lcl"         /* lokale Variable        */
#define MOD_IMMED   "con"         /* "load": Constant       */

/*
 *      OP_ALU Namen
 */

#define ALU_ADD "+"               /* Addition               */
#define ALU_SUB "-"               /* Subtraktion            */
#define ALU_MUL "*"               /* Multiplikation         */
#define ALU_DIV "/"               /* Division               */
#define ALU_MOD "%"               /* Rest                   */
```

```
#define ALU_LT   "<"            /* Vergleich:   <        */
#define ALU_GT   ">"            /*              >        */
#define ALU_LE   "<="           /*              <=       */
#define ALU_GE   ">="           /*              >=       */
#define ALU_EQ   "=="           /*              ==       */
#define ALU_NE   "!="           /*              !=       */
#define ALU_AND  "&"            /* Bit-UND               */
#define ALU_OR   "|"            /* Bit-ODER              */
#define ALU_XOR  "^"            /* Bit-ODER exklusiv     */

/*
 *      Typvereinbarungen fuer Funktionen
 */

char * gen_mod();               /* Bereichsname */
```

Die Datei *gen.c* enthält die C-Funktionen, die Assembler-Text für unsere fiktive *kleinC*-Maschine entwickeln. Diese Funktionen wurden im Kapitel 7 erklärt.

gen.c

```
/*
 *      kleinC -- Code-Generierung
 */

#include "symtab.h"
#include "gen.h"

/*
 *      verschiedene Befehlsformate generieren
 */

gen_alu(mod, comment)
        char * mod;             /* ALU Operationsname */
        char * comment;         /* Kommentar */
{
        printf("\t%s\t%s\t\t; %s\n", OP_ALU, mod, comment);
}

gen_li(const)
        char * const;           /* Constant: Wert */
{
        printf("\t%s\t%s,%s\n", OP_LOAD, MOD_IMMED, const);
}
```

```
char * gen_mod(symbol)
        struct symtab * symbol;
{
        switch (symbol->s_blknum) {
        case 1:
                return MOD_GLOBAL;
        case 2:
                return MOD_PARAM;
        }
        return MOD_LOCAL;
}

gen(op, mod, val, comment)
        char * op;              /* Befehlsname */
        char * mod;             /* Bereichsname */
        int val;                /* Offset */
        char * comment;         /* Kommentar */
{
        printf("\t%s\t%s,%d\t\t; %s\n", op, mod, val, comment);
}

gen_pr(op, comment)
        char * op;              /* Befehlsname */
        char * comment;         /* Kommentar */
{
        printf("\t%s\t\t\t; %s\n", op, comment);
}

/*
 *      interne Marke druckbar darstellen
 */

#define LABEL   "$$%d"

static char * format_label(label)
        int label;
{       static char buffer[sizeof LABEL + 2];

        sprintf(buffer, LABEL, label);
        return buffer;
}
```

```
/*
 *      Sprungbefehl generieren; Resultat ist Zielmarke
 */

int gen_jump(op, label, comment)
        char * op;              /* Befehlsname */
        int label;              /* Sprungziel */
        char * comment;         /* Kommentar */
{
        printf("\t%s\t%s\t\t; %s\n", op, format_label(label),
                comment);
        return label;
}

/*
 *      eindeutige interne Marke erzeugen
 */

int new_label()
{       static int next_label = 0;

        return ++next_label;
}

/*
 *      interne Marke definieren
 */

int gen_label(label)
        int label;
{
        printf("%s\tequ\t*\n", format_label(label));
        return label;
}

/*
 *      Marken-Stack Manager
 */

static struct bc_stack {
        int bc_label;           /* Marke (von new_label()) */
        struct bc_stack * bc_next;
        } * b_top,              /* Beginn BREAK-Stack */
          * c_top;              /* Beginn CONTINUE-Stack */
```

```
static struct bc_stack * push(stack, label)
        struct bc_stack * stack;
        int label;
{       struct bc_stack * new_entry = (struct bc_stack *)
                calloc(1, sizeof(struct bc_stack));

        if (new_entry)
        {       new_entry->bc_next = stack;
                new_entry->bc_label = label;
                return new_entry;
        }
        fatal("Kein Platz mehr fuer Schleifen.");
        /*NOTREACHED*/
}

static struct bc_stack * pop(stack)
        struct bc_stack * stack;
{       struct bc_stack * old_entry;

        if (stack)
        {       old_entry = stack;
                stack = old_entry->bc_next;
                cfree(old_entry);
                return stack;
        }
        bug("break/continue Stack ist leer");
        /*NOTREACHED*/
}

static int top(stack)
        struct bc_stack * stack;
{
        if (! stack)
        {       error("keine Schleife offen");
                return 0;
        }
        else
                return stack->bc_label;
}
```

```
/*
 *      BREAK und CONTINUE
 */

push_break(label)
        int label;
{
        b_top = push(b_top, label);
}

push_continue(label)
        int label;
{
        c_top = push(c_top, label);
}

pop_break()
{
        b_top = pop(b_top);
}

pop_continue()
{
        c_top = pop(c_top);
}

gen_break()
{
        gen_jump(OP_JUMP, top(b_top), "BREAK");
}

gen_continue()
{
        gen_jump(OP_JUMP, top(c_top), "CONTINUE");
}
```

```
/*
 *      Funktionsaufruf
 */

gen_call(symbol, count)
        struct symtab * symbol; /* Funktion */
        int count;              /* Anzahl Argumente */
{
        chk_parm(symbol, count);
        printf("\t%s\t%d,%s\n", OP_CALL, count, symbol->s_name);
        while (count-- > 0)
                gen_pr(OP_POP, "Argument entfernen");
        gen(OP_LOAD, MOD_GLOBAL, 0, "Resultat auf Stack");
}

/*
 *      Befehle am Beginn einer Funktion
 */

int gen_entry(symbol)
        struct symtab * symbol; /* Funktion */
{       int label = new_label();

        printf("%s\t", symbol->s_name);
        printf("%s\t%s\n", OP_ENTRY, format_label(label));
        return label;
}

fix_entry(symbol, label)
        struct symtab * symbol; /* Funktion */
        int label;
{       extern int l_max;       /* Groesse, lokaler Bereich */

        printf("%s\tequ\t%d\t\t; %s\n", format_label(label),
                l_max, symbol->s_name);
}
```

```
/*
 *      Abschluss eines Programms
 */

end_program()
{       extern int g_offset;      /* Groesse, globaler Bereich */

        all_program();            /* globale Variablen anlegen */
        printf("\tend\t%d,main\n", g_offset);
}
```

Ein Compiler mit Code-Generator für unsere fiktive Maschine kann mit folgendem Kommando erzeugt werden:

```
cc gen.c mem.c symtab.c message.c y.tab.c lex.yy.c \
        colib -ll -o gen
```

colib enthält gewisse Hilfsfunktionen für *yacc*, die im Abschnitt A.8 beschrieben werden.

A.7 Simulation

Die Definitionsdatei *sim.h* ersetzt *gen.h* innerhalb von *kleinc.y*, wenn ein Compiler mit Simulator konstruiert wird. In beiden Definitionsdateien werden die gleichen Namen vereinbart, aber *sim.h* enthält die für den Simulator nötigen Definitionen.

sim.h

```
/*
 *      kleinC -- Definitionsdatei fuer Simulation
 */

/*
 *      Befehlsnamen fuer fiktive Maschine
 */

#define OP_ALU    1     /* alu     Arithmetik-Logik-Op  */
#define OP_DEC    2     /* dec     Bereich,Offset       */
#define OP_INC    3     /* inc     Bereich,Offset       */
#define OP_LOAD   4     /* load    Bereich,Offset       */
#define OP_STORE  5     /* store   Bereich,Offset       */
#define OP_POP    6     /* pop                          */
#define OP_JUMPZ  7     /* jumpz   Marke                */
#define OP_JUMP   8     /* jump    Marke                */
#define OP_CALL   9     /* call    Funktionsadresse     */
#define OP_ENTRY  10    /* entry   lokaler-Bereich      */
#define OP_RETURN 11    /* return                       */
```

```
/*
 *      Bereichsnamen
 */

#define MOD_GLOBAL 1    /* globale Variable     */
#define MOD_PARAM  2    /* Parameter            */
#define MOD_LOCAL  3    /* lokale Variable      */
#define MOD_IMMED  4    /* "load": Constant     */

/*
 *      OP_ALU Namen
 */

#define ALU_ADD 1       /* Addition             */
#define ALU_SUB 2       /* Subtraktion          */
#define ALU_MUL 3       /* Multiplikation       */
#define ALU_DIV 4       /* Division             */
#define ALU_MOD 5       /* Rest                 */
#define ALU_LT  6       /* Vergleich:   <       */
#define ALU_GT  7       /*              >       */
#define ALU_LE  8       /*              <=      */
#define ALU_GE  9       /*              >=      */
#define ALU_EQ  10      /*              ==      */
#define ALU_NE  11      /*              !=      */
#define ALU_AND 12      /* Bit-UND              */
#define ALU_OR  13      /* Bit-ODER             */
#define ALU_XOR 14      /* Bit-ODER exklusiv    */

/*
 *      Programm-Speicher
 */

struct prog {
        short p_op;     /* Befehlsname */
        short p_mod;    /* Bereich/ALU Name */
        int p_val;      /* Offset o. ae. */
        };

/*
 *      aenderbare Groessen
 */

#define L_PROG  200     /* max. Programmgroesse */
#define L_DATA  100     /* max. fuer Stack und Variablen */
```

```
/*
 *      Ausdehnung eines Vektors
 */

#define DIM(x)  (sizeof x / sizeof x[0])
```

Die Datei *sim.c* enthält den Simulator für unsere fiktive Maschine, der im Kapitel 8 erarbeitet wurde.

sim.c

```
/*
 *      kleinC -- Simulator
 */

#include "sim.h"

/*
 *      Daten- und Programmspeicher
 */

static int data[L_DATA];
extern struct prog prog[];

/*
 *      Register
 */

static struct prog * inst;  /* -> aktueller Befehl */
#define G       0           /* globaler Bereich */
static int P;               /* aktueller Parameterbereich */
static int L;               /* aktueller lokaler Bereich */
static int T;               /* Stack-Spitze */

/*
 *      Abkuerzungen
 */

#define TOP     data[T-1]   /* rechter Operand: Stack-Spitze */
#define NEXT    data[T-2]   /* linker Operand: unter TOP */
#define PUSH    data[T++]   /* neuer Eintrag auf den Stack */
#define POP     -- T        /* -> Eintrag, vom Stack entfernt */
#define MEMORY  data[address()]
                            /* effektiv adressiertes Wort */
#define RESULT  data[G]     /* Resultatwert einer Funktion */
```

```
/*
 *      Adress-Decodierung
 */

static int address()    /* effektive Datenadresse */
{       register int ad;

        switch (inst->p_mod) {
        case MOD_GLOBAL:
                ad = G;
                break;
        case MOD_PARAM:
                ad = P;
                break;
        case MOD_LOCAL:
                ad = L;
                break;
        default:
                bug("p_mod ist falsch");
        }
        ad += inst->p_val;
        if (ad < 0 || ad >= T)
                bug("effektive Adresse ist falsch");
        return ad;
}

/*
 *      Simulator
 */

simulate(pc_limit, global, pc)
        int pc_limit, global, pc;
{
        /* initialisieren */
        if (global >= DIM(data))
           fatal(
              "Nicht genuegend Platz fuer globale Variablen.");
        T = global + 2;

        printf("\nAusfuehrung beginnt...\n\n");
```

```
for (;;)
{       /* fetch */
        if (pc < 0 || pc >= pc_limit)
                bug("pc nicht im Programm");
        inst = &prog[pc++];

        /* Operation decodieren */
        switch (inst->p_op) {
        default:
                printf("%d:\thalt\n", inst-prog);
                return;
        case OP_ALU:
                if (T <= L+1)
                        bug("Simulator-Stack leer");
                switch (inst->p_mod) {
                default:
                        bug("illegaler ALU-Befehl");
                case ALU_ADD: NEXT += TOP; break;
                case ALU_SUB: NEXT -= TOP; break;
                case ALU_MUL: NEXT *= TOP; break;
                case ALU_DIV: NEXT /= TOP; break;
                case ALU_MOD: NEXT %= TOP; break;
                case ALU_LT:  NEXT = NEXT < TOP; break;
                case ALU_GT:  NEXT = NEXT > TOP; break;
                case ALU_LE:  NEXT = NEXT <= TOP; break;
                case ALU_GE:  NEXT = NEXT >= TOP; break;
                case ALU_EQ:  NEXT = NEXT == TOP; break;
                case ALU_NE:  NEXT = NEXT != TOP; break;
                case ALU_AND: NEXT &= TOP; break;
                case ALU_OR:  NEXT |= TOP; break;
                case ALU_XOR: NEXT ^= TOP; break;
                }
                POP;
                break;
        case OP_LOAD:
                if (T >= DIM(data))
                        fatal("Zu viele Werte.");
                if (inst->p_mod == MOD_IMMED)
                        PUSH = inst->p_val;
                else
                        PUSH = MEMORY;
                break;
```

```
case OP_STORE:
        if (T <= L)
                bug("Simulator Stack ist leer");
        printf("%d:\tstore\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
                inst->p_val, MEMORY = TOP);
        break;
case OP_INC:
        if (T >= DIM(data))
                fatal("Zu viele Werte.");
        printf("%d:\tinc\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
                inst->p_val, PUSH = ++ MEMORY);
        break;

case OP_DEC:
        if (T >= DIM(data))
                fatal("Zu viele Werte.");
        printf("%d:\tdec\t%d,%d\tto %d\n",
                inst-prog, inst->p_mod,
                inst->p_val, PUSH = -- MEMORY);
        break;
case OP_POP:
        if (T <= L)
                bug("Simulator Stack ist leer");
        POP;
        break;
case OP_JUMP:
        printf("%d:\tjump\t%d\n", inst-prog,
                inst->p_val);
        pc = inst->p_val;
        break;
case OP_JUMPZ:
        if (T <= L)
                bug("Simulator Stack ist leer");
        if (data[POP] == 0)
        {       printf("%d:\tjumpz\t%d\n",
                        inst-prog, inst->p_val);
                pc = inst->p_val;
        }
        break;
```

```
            case OP_CALL:
                    printf("%d:\tcall\t%d\n", inst-prog,
                            inst->p_val);
                    PUSH = pc;
                    pc = inst->p_val;
                    PUSH = P;
                    P = T - 2 - inst->p_mod;
                    break;
            case OP_ENTRY:
                    PUSH = L;
                    L = T;
                    T += inst->p_val;
                    if (T >= DIM(data))
                            fatal("Zu viele Werte.");
                    break;
            case OP_RETURN:
                    if (T < L)
                            bug("Simulator Stack ist leer");
                    T = L;
                    L = data[POP];
                    P = data[POP];
                    pc = data[POP];
                    printf("%d:\treturn\t%d to %d\n",
                            inst-prog, RESULT, pc);
                    break;
            }
    }
```

Die Funktionen zur Code-Generierung in der Datei *simgen.c* treten an die Stelle der entsprechenden Funktionen in der Datei *gen.c* im Abschnitt A.6.

simgen.c

```
/*
 *      kleinC -- Code-Generierung fuer den Simulator
 */

#include "sim.h"
#include "symtab.h"
```

```
/*
 *      Programmspeicher
 */

struct prog prog[L_PROG];
static int pc = 1;              /* aktueller Programmzaehler */
                                /* HALT (0) ist bei Adresse 0 */

/*
 *      einen Befehl im Programmspeicher ablegen
 */

int gen(op, mod, val, comment)
        int op;                 /* Befehlsname */
        int mod;                /* Bereich/ALU Name */
        int val;                /* Offset o.ae. */
        char * comment;         /* Kommentar */
{
        if (pc >= DIM(prog))
                fatal("Kein Platz mehr im Programmspeicher.");
        prog[pc].p_op = op;
        prog[pc].p_mod = mod;
        prog[pc].p_val = val;
        printf("%d:\t%d\t%d,%d\t; %s\n",
                pc, op, mod, val, comment);
        return pc ++;
}

/*
 *      Bereichsname
 */

int gen_mod(symbol)
        struct symtab * symbol;
{
        switch (symbol->s_blknum) {
        case 1:
                return MOD_GLOBAL;
        case 2:
                return MOD_PARAM;
        }
        return MOD_LOCAL;
}
```

```
/*
 *      allgemeine Maschinenbefehle
 */

gen_alu(mod, comment)
        int mod;                /* ALU Name */
        char * comment;         /* Kommentar */
{
        gen(OP_ALU, mod, 0, comment);
}

gen_li(const)
        char * const;           /* Constant: Wert */
{
        gen(OP_LOAD, MOD_IMMED, atoi(const), const);
}

gen_pr(op, comment)
        int op;                 /* Befehlsname */
        char * comment;         /* Kommentar */
{
        gen(op, 0, 0, comment);
}

/*
 *      Sprungbefehl generieren;
 *      Resultat ist Zieladresse oder Kette
 */

int gen_jump(op, label, comment)
        int op;                 /* Befehlsname */
        int label;              /* Sprungziel */
        char * comment;         /* Kommentar */
{       int pc = gen(op, 0, label, comment);

        if (label <= 0)
                return -pc;     /* neues Ende der Kette */
        else
                return label;   /* bereits definiert */
}
```

```
/*
 *      Ende einer Kette von Vorwaerts-Verweisen
 */

int new_label()
{
        return 0;               /* Anfang der Kette */
}

/*
 *      Vorwaerts-Verweise reparieren
 */

int gen_label(chain)
        int chain;
{       int next;

        while (chain < 0)
        {       chain = - chain;
                next = prog[chain].p_val;
                if (next > 0)
                        break;  /* schon repariert */
                prog[chain].p_val = pc;
                printf("%d:\t(korr.)\t%d\n", chain, pc);
                chain = next;
        }
        return pc;
}

/*
 *      Marken-Stack Manager
 */

static struct bc_stack {
        int bc_label;           /* Marke (von new_label()) */
        struct bc_stack * bc_next;
        } * b_top,              /* Beginn BREAK-Stack */
          * c_top;              /* Beginn CONTINUE-Stack */
```

```
static struct bc_stack * push(stack, label)
        struct bc_stack * stack;
        int label;
{       struct bc_stack * new_entry = (struct bc_stack *)
                calloc(1, sizeof(struct bc_stack));

        if (new_entry)
        {       new_entry->bc_next = stack;
                new_entry->bc_label = label;
                return new_entry;
        }
        fatal("Kein Platz mehr fuer Schleifen.");
        /*NOTREACHED*/
}

static struct bc_stack * pop(stack)
        struct bc_stack * stack;
{       struct bc_stack * old_entry;

        if (stack)
        {       old_entry = stack;
                stack = old_entry->bc_next;
                cfree(old_entry);
                return stack;
        }
        bug("break/continue Stack ist leer");
        /*NOTREACHED*/
}

static int * top(stack)
        struct bc_stack ** stack;
{
        if (! *stack)
        {       error("keine Schleife offen");
                *stack = push(*stack, 0);
        }
        return & (*stack) -> bc_label;
}
```

```
/*
 *      BREAK und CONTINUE
 */

push_continue(label)
        int label;
{
        c_top = push(c_top, label);
}

push_break(label)
        int label;
{
        b_top = push(b_top, label);
}

gen_break()
{
        *top(&b_top) = gen_jump(OP_JUMP, *top(&b_top),
                                "BREAK");
}

gen_continue()
{
        *top(&c_top) = gen_jump(OP_JUMP, *top(&c_top),
                                "CONTINUE");
}

pop_break()
{
        gen_label(*top(&b_top));
        b_top = pop(b_top);
}

pop_continue()
{
        gen_label(*top(&c_top));
        c_top = pop(c_top);
}
```

```
/*
 *      Funktionsaufruf
 */

gen_call(symbol, count)
        struct symtab * symbol; /* Funktion */
        int     count;          /* Anzahl Argumente */
{       int pc;

        chk_parm(symbol, count);
        pc = gen(OP_CALL, count, symbol->s_offset,
                symbol->s_name);
        if (symbol->s_offset <= 0)
                symbol->s_offset = -pc; /* Ende der Kette */
        while (count-- > 0)
                gen_pr(OP_POP, "Argument entfernen");
        gen(OP_LOAD, MOD_GLOBAL, 0, "Resultat auf Stack");
}

/*
 *      Code am Beginn einer Funktion, Definition der Funktion
 */

int gen_entry(symbol)
        struct symtab * symbol; /* Funktion */
{
        symbol->s_offset = gen_label(symbol->s_offset);
        gen(OP_ENTRY, 0, 0, symbol->s_name);
        return symbol->s_offset;
}

fix_entry(symbol, label)
        struct symtab * symbol; /* Funktion */
        int label;
{       extern int l_max;    /* Groesse des lokalen Bereichs */

        prog[label].p_val = l_max;
        printf("%d:\tentry\t%d\t; %s\n",
                label, l_max, symbol->s_name);
}
```

```
/*
 *      Abschluss eines Programms
 */
end_program()
{       extern int g_offset; /* Groesse des globalen Bereichs */
        extern struct symtab * s_find();
        int main = s_find("main") -> s_offset;

        all_program();          /* globale Variablen anlegen */
        printf("%d:\tend\t%d,%d\n", pc, g_offset, main);
        simulate(pc, g_offset, main);
}
```

Ein Compiler mit Simulator für unsere fiktive Maschine kann mit folgendem Kommando erzeugt werden:

```
cc sim.c simgen.c mem.c symtab.c message.c y.tab.c lex.yy.c \
        colib -ll -o sim
```

A.8 Bessere Fehlermeldungen aus "yaccpar"

Die Fehlermeldungen, die ein von *yacc* erzeugter Parser ausgibt, kann man verbessern. *yacc* entwickelt den Parser aus einem Modell, das sich in der Datei */usr/lib/yaccpar* befindet. Normalerweise passiert in *yaccpar* bei Erkennung eines Syntaxfehlers folgendes:

```
switch (yyerrflag) {
case 0:                         /* brand new error */
        yyerror("syntax error");
yyerrlab:
        ++yynerrs;
case 1:                         /* incompletely recovered */
case 2:                         /* ... try again */
        yyerrflag = 3;
        ...
```

Der **switch** wird erreicht, wenn die Übergangsmatrix eine **error**-Operation verlangt. In diesem Fall wird also schlicht die Meldung **syntax error** ausgegeben, ohne Rücksicht auf den tatsächlichen Fehler.

yyerrflag verhindert einen Schwall von Fehlermeldungen. Die Variable ist normalerweise null. Wird ein Syntaxfehler erkannt, erhält **yyerrflag** den Wert drei. Bei jeder **shift**-Operation wird **yyerrflag** dann wieder dekrementiert. Eine Fehlermeldung erfolgt nur, wenn **yyerrflag** den Wert null hat.[1]

[1] Die **yyerrok**-Aktion setzt gerade **yyerrflag = 0**. A.d.Ü.

Wir können folgendermaßen versuchen, die Fehlermeldung zu verbessern: im momentanen Zustand des Parsers sollten wir *alle möglichen* Eingabesymbole an Stelle der tatsächlichen Eingabe betrachten, um die Eingabesymbole zu finden, für die die Übergangsmatrix keine **error**-Operation enthält. Entsprechend formatiert, kann man dann eine Liste dieser Symbole an Stelle der lakonischen **syntax error** Fehlermeldung ausgeben.

/usr/lib/yaccpar ist immer als Quelle vorhanden. Wir berechnen die gewünschte Symbolliste im wesentlichen, indem wir den in *yaccpar* für ein Symbol aus der Eingabe verwendeten Algorithmus kopieren:

```
switch (yyerrflag) {
case 0:                              /* brand new error */
        if ((yyn = yypact[yystate]) > YYFLAG && yyn < YYLAST)
        {       register int x;

                for (x = yyn>0? yyn: 0; x < YYLAST; ++x)
                        if (yychk[yyact[x]] == x - yyn
                            && x - yyn != YYERRCODE)
                                yyerror(0, yydisplay(x-yyn));
        }
        yyerror(0,0);
yyerrlab:
        ++yynerrs;
        ...
```

yystate ist der momentane Parser-Zustand, **YYLAST** ist die Größe der als **yyact[]** codierten Übergangsmatrix, und **yyn** ist eine temporäre Variable für *yaccpar*. Der Vektor **yypact[]** dient als Filter, mit dem bestimmte einfache Fälle ausgeschlossen werden. Die Vektoren werden von *yacc* initialisiert.

Wenn **yyact[]** wirklich betrachtet wird, durchlaufen wir mit **x** alle möglichen Indexwerte für **yyact[]**. Ein Indexwert kann nicht negativ sein, und der Algorithmus in *yaccpar* zeigt, daß der Indexwert auch größer als **yypact[yystate]** sein muß. Entdecken wir einen Indexwert **x**, der nicht dem Fehlersymbol **YYERRCODE** entspricht, und der zu keiner **error**-Operation in **yyact[]** führt, dann geben wir das zugehörige Eingabesymbol aus.

Kann auf diese Weise eine Liste von möglichen Eingabesymbolen berechnet werden, dann ruft diese neue Version von *yaccpar*

```
yyerror(0,t)
```

für jedes Eingabesymbol **t** in der Liste auf; **t** wird dabei als Zeichenkette repräsentiert. Als Abschluß der Liste erfolgt ein Aufruf

```
yyerror(0,0)
```

Dieser Aufruf erfolgt auch, wenn die Liste nicht berechnet werden kann (oder leer ist). **yyerror()** kann folgendermaßen erweitert werden, um diese verbesserten Fehlermeldungen zu liefern:

```
/*
 *      yyerror() -- [verbesserte] Fehlermeldung fuer yyparse()
 */

#include <stdio.h>

FILE * yyerfp = stdout;          /* alle Meldungen */

                                 /*VARARGS1*/
yyerror(s,t)                     /* "text" oder 0,"symbol" */
        register char * s, * t;
{       extern int yynerrs;      /* Anzahl Fehler */
        static int list = 0;     /* != 0 in Symbolliste */

        if (s || ! list)         /* Kopf noetig?? */
        {       fprintf(yyerfp, "[Fehler %d] ", yynerrs+1);
                yywhere();
                if (s)           /* einfache Meldung?? */
                {       fputs(s, yyerfp);
                        putc('\n', yyerfp);
                        return;
                }
                if (t)           /* erstes Symbol?? */
                {       fputs("erwartet wird: ", yyerfp);
                        fputs(t, yyerfp);
                        list = 1;
                        return;
                }
                                 /* kein Symbol moeglich */
                fputs("Syntax Fehler\n", yyerfp);
                return;
        }
        if (t)                   /* weiteres Symbol?? */
        {       putc(' ', yyerfp);
                fputs(t, yyerfp);
                return;
        }
                        /* Ende der verbesserten Meldung */
        putc('\n', yyerfp);
        list = 0;
}
```

Die Implementierung der neuen Fehlermeldung (**erwartet wird:** *symbole* in **yyerror()**) ist recht einfach. Im Prinzip bearbeiten wir die verschiedenen Aufrufe abhängig davon, ob wir Nullzeiger erkennen. Die **static** vereinbarte Variable **list** behält ihren Wert über alle Aufrufe hinweg; sie dient dazu, das erste Eingabesymbol in einer Liste von den nachfolgenden Symbolen zu unterscheiden, damit die Fehlermeldung entsprechend eingeleitet werden kann.

Eingabesymbole werden während der lexikalischen Analyse als Integer-Werte codiert. In *yaccpar* verwenden wir eine Funktion **yydisplay()**, die diese Codierung symbolisch druckbar darstellt:

```
#include <ctype.h>
#define DIM(x)  (sizeof x / sizeof x[0])

static char * yydisplay(ch)
        register int ch;
{       static char buf[15];
        static char * token[] = {
#include "y.tok.h"              /* %token Namen */
                0 };

        switch(ch) {
        case 0:
                return "[Dateiende]";
        case YYERRCODE:
                return "[Fehler]";
        case '\b':
                return "'\\b'";
        case '\f':
                return "'\\f'";
        case '\n':
                return "'\\n'";
        case '\r':
                return "'\\r'";
        case '\t':
                return "'\\t'";
        }
        if (ch > 256 && ch < 256 + DIM(token))
                return token[ch - 257];
        if (isascii(ch) && isprint(ch))
                sprintf(buf, "'%c'", ch);
        else if (ch < 256)
                sprintf(buf, "char %04.3o", ch);
        else
                sprintf(buf, "Symbol %d", ch);
        return buf;
}
```

Falls das Eingabesymbol **ch** ein druckbares ASCII-Zeichen ist, oder falls eine Ersatzdarstellung mit \ existiert, wird das Symbol im Stil einer C-Zeichenkonstanten dargestellt. Ist der Wert des Eingabesymbols größer als 256, so wurde er normalerweise von *yacc* aufgrund einer **%token**-Anweisung definiert; in diesem Fall versuchen wir, den Wert mit Hilfe eines Vektors **token[]** zu decodieren, der geeignete Zeichenketten enthalten sollte. Funktioniert keine dieser Methoden, so stellen wir den entsprechenden numerischen Wert selbst dar. Die Definition für **yydisplay()** sollte man relativ weit vorn in der Datei */usr/lib/yaccpar* einfügen; mit der Funktion kann man auch die Ausgabe bei der Ablaufverfolgung durch Definition von **YYDEBUG** verbessern.

Die Initialisierung des **token[]**-Vektors kann man mit dem folgenden Skript für die Bourne-Shell direkt aus der Definitionsdatei *y.tab.h* entwickeln:

```
grep '^# define' y.tab.h \
| sed 's/^# define \([^ ]*\) [^ ]*$/    "\1",/'  > y.tok.h
```

Die Zeichenketten werden in einer Datei *y.tok.h* abgelegt. Damit das Verfahren auch dann funktioniert, wenn diese Datei lokal nicht existiert, sollte man eine leere Datei */usr/include/y.tok.h* im System anlegen, die dann in **yydisplay()** als 'Notbremse' eingefügt wird.

Die hier gezeigte neue Version von **yyerror()** und die Funktionen, die im Abschnitt 3.3 besprochen wurden, stellt man am besten in einer Bücherei *colib* zusammen:

```
cc -c main.c cpp.c yywhere.c yyerror.c
ar r colib main.o cpp.o yywhere.o yyerror.o
ranlib colib
```

A.9 Bessere Ablaufverfolgung von "yaccpar"

Auch die Ausgabe zur Ablaufverfolgung in */usr/lib/yaccpar* kann verbessert werden. Ziemlich ärgerlich ist nämlich, daß manche Eingabesymbole verwendet und weggeworfen werden, *bevor* sie in der Ablaufverfolgung gezeigt werden. *yaccpar* macht in diesem Zusammenhang folgendes:

```
#ifdef  YYDEBUG
int yydebug = 0;
#endif
int yychar;     /* current input token number */

yyparse()
{       register short yystate;
        ...
        yychar = -1;
```

```
yystack:        /* put a state and value onto the stack */

#ifdef  YYDEBUG
        if (yydebug)
                printf("state %d, char 0%o\n", yystate, yychar);
#endif

        ...
        if (yychar < 0)         /* lookahead available? */
                if ((yychar = yylex()) < 0)
                        yychar = 0;     /* end of file */
        ...
        if ( /* valid shift */ )
        {       ...
                yychar = -1;
                goto yystack;
        }
        ...
```

Um die Ablaufverfolgung zu verbessern, müssen wir jedes Eingabesymbol darstellen unmittelbar nachdem es von **yylex()** geliefert wird. Da die oben gezeigten, verschachtelten **if**-Anweisungen zweimal in *yaccpar* vorkommen, ersetzen wir sie am besten durch den Aufruf einer neuen Funktion **yyyylex()**, die wir folgendermaßen definieren:

```
static yyyylex()
{
        if (yychar < 0)
        {       if ((yychar = yylex ()) < 0)
                        yychar = 0;
#ifdef YYDEBUG
        if (yydebug)
                printf("[yydebug] Eingabe %s\n",
                        yydisplay(yychar));
#endif
        }
}
```

wobei wir **yydisplay()** verwenden, um eine relativ verständliche Darstellung des Eingabesymbols zu erhalten.

A.10 Systematische Tests

Eingabedateien, die zum Test eines Compilers in der Entwicklungsphase verwendet werden, sollte man aufbewahren. Ändert man dann den Compiler, so kann man diese Dateien nochmals übersetzen um sicherzustellen, daß wenigstens einige vorher funktionsfähige Teile des Compilers noch unverändert arbeiten. Das folgende Skript für die Bourne Shell automatisiert dieses Vorgehen:

```
if test ! -d $OD
then    mkdir $OD
fi

for i in $ID/*
do      file=`basename $i`
        "$@" $i >$OD/\&$file
        echo 'exit() Code' $? >>$OD/\&$file
        if test -r $OD/$file
        then    if cmp -s $OD/\&$file $OD/$file
                then    echo "$1 $file unveraendert"
                        rm $OD/\&$file
                else    echo "$1 $file veraendert: <neu >alt"
                        diff $OD/\&$file $OD/$file
                        mv $OD/$file $OD/${file}_old
                        mv $OD/\&$file $OD/$file
                fi
        else    echo "$1 $file erzeugt"
                mv $OD/\&$file $OD/$file
        fi
done
exit 0
```

$ID muß als Name des Katalogs definiert werden, der die Eingabedateien enthält. **$OD** ist analog der Katalog, in dem für jede Eingabedatei eine entsprechende Ausgabedatei angelegt wird. Ein- und Ausgabedatei haben dabei in den beiden Katalogen die gleichen Namen. Temporäre Dateinamen beginnen mit **&**. Unterscheiden sich neue und vorhergehende Version einer Ausgabedatei, so wird die vorhergehende Version mit der Nachsilbe **_old** umbenannt und aufbewahrt.

Das Shell-Skript erwartet als Argument den Namen des Prozessors, der geprüft werden soll. Optional können weitere Argumente angegeben werden, die an den Prozessor vor dem Namen der Eingabedatei übergeben werden. Der Reihe nach wird dann jede Eingabedatei bearbeitet, und die Ausgabe wird mit der vorhergehenden Version verglichen. Sind beide Versionen gleich, wird die *alte* Version aufbewahrt (damit ihr Zeitstempel in der Dateihierarchie erhalten bleibt). Sind die Versionen verschieden, werden die Unterschiede mit *diff* angezeigt. Existierte noch keine Ausgabe, wird dies entsprechend berichtet.

Das Skript verlangt, daß der Prozessor einen Dateinamen für die Eingabedatei als Argument verarbeiten kann. Dies ist nicht der Fall bei einem von *lex* generierten Programm, das die **main()**-Funktion aus der *lex*-Bücherei verwendet. Für solche Fälle kann man den Aufruf des Prozessors entsprechend modifizieren:

```
"$@" <$i >$OD/\&$file
```

"$@" muß man als Aufruf des Prozessors verwenden, damit jedes Argument des Shell-Skripts unverändert auch dann an den Prozessor als einzelnes Argument übergeben wird, wenn es Zwischenraumzeichen enthält.

Literaturverzeichnis

[Aho74] A. V. Aho and S. C. Johnson, *LR Parsing,* Comp. Surveys, vol. 6, no. 2, pp. 99-124, June 1974.

[Aho77] A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977.

[Bau76] F. L. Bauer and J. Eickel (ed.), *Compiler Construction: An Advanced Course,* Springer, Berlin, 1974, 1976.

[Gra79] S. L. Graham, C. B. Haley, and W. N. Joy, *Practical LR error recovery,* SIGPLAN Notices, Aug 1979.

[Gri71] D. Gries, *Compiler Construction for Digital Computers,* Wiley, New York, 1971.

[Jen75] K. Jensen and N. Wirth, *Pascal: User Manual and Report,* Springer, Berlin, 1975.

[Joh78] S. C. Johnson, *Yacc: Yet Another Compiler-Compiler,* in [Ker78a].

[Ker78a] B. W. Kernighan and M. D. McIlroy, *UNIX Programmer's Manual,* Bell Laboratories, 1978. Seventh Edition.

[Ker78b] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978. Deutsche Ausgabe: *Programmieren in C,* Carl Hanser Verlag, München Wien 1983.

[Les78a] M. E. Lesk, *Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff,* in [Ker78a].

[Les78b] M. E. Lesk and E. Schmidt, *Lex: A Lexical Analyzer Generator,* in [Ker78a].

[Nau63] P. Naur (ed.), *Revised Report on the Algorithmic Language Algol 60,* CACM, pp. 1-17, Jan. 1963.

[Sch85] A. T. Schreiner, *UNIX Sprechstunde,* unix/mail, no. 1, 1985.

[Wic73] B. A. Wichman, *The Definition of Comments in Programming Languages,* NPL Report NAC-34, National Physics Laboratory, Division of Numerical Analysis and Computing, Teddington, England, May 1973.

[Wij75] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, *Revised Report on the Algorithmic Language Algol 68,* Acta Informatica, pp. 1-236, 1, 1975.

[Wir77] N. Wirth, *Compilerbau,* B. G. Teubner, Stuttgart, 1977.

[Wir82] N. Wirth, *Programming in Modula-2,* Springer, Berlin, 1982.
gn, Addison-Wesley, Reading, Mass., 1977.

%left 21, 25, 55
%nonassoc 21, 55
%prec 82
%right 21, 25, 55
%token 15, 24, 45, 54, 55, 136
%type 136
%union 135
accept-Operation (in *yacc*) 52
activation record 139
Aktion (in *yacc*) 70
Aktion innerhalb einer Formulierung (in *yacc*) 83
anonyme Aktion (in *yacc*) 83
Assembler 5, 151, 215
assert() 44
Assoziativität 12, 21
Automat, endlicher 31
Automat, *push-down* 52
Backus Naur Form (BNF) 6, 7, 14, 24, 29
blindes Element (in Symboltabelle) 118
break-Stack 152, 154, 161, 184, 185
Bücherei 40, 239
Bücherei (in *lex*) 40
Bücherei (in *yacc*) 239
bug() 133, 202
C Positionsangabe, Muster 45, 49
C-Preprozessor 45, 57, 58, 72
C-Preprozessor Positionsangaben 49, 61
compound statement 138, 144, 157
continue-Stack 152, 154, 161, 184, 185
cpp() 58
Definition 2
Definition von Funktionen 126
Definition von Variablen 124
Deklaration 2, 109, 121, 124, 126
Deklaration von Funktionen 114
Deklaration von Parametern 114
deterministische Grammatik 13
display 140
Eingabesymbol 7, 11, 13, 31, 69
Eingabesymbol, Repräsentierung 54
Eingabesyntax (in *lex*) 39
Eingabesyntax (in *yacc*) 21
else-Problem 19, 100
endlicher Automat 31
error() 133, 202
error-Operation (in *yacc*) 52, 90
error-Symbol (in *yacc*) 89, 90
error-Symbol, Plazierung (in *yacc*) 93, 94
Extended Backus Naur Form (EBNF) 10, 22, 29
fatal() 133, 202
Fehlermeldung 56, 60, 202, 235
Fehlersuche 44, 45, 48, 55, 62, 116, 239, 241
Fehlersuche (in *lex*) 44, 45, 48
Fehlersuche (in *yacc*) 62, 239
Formulierung 2, 7, 13
Funktionsaufruf 151, 154, 163, 176, 177, 185
Funktionsdefinition 126
Funktionsdeklaration 114
Funktionsende 154, 164, 177
Geltungsbereich (eines Namens) 109, 138
Gleitkommakonstante, Muster 73
goto-Operation (in *yacc*) 53, 70
Grammatik 7, 8, 11, 14
Grammatik, deterministische 13
Grammatik, Eingabesymbol 7, 11, 13, 31, 69
Grammatik, Grammatikbegriff 7, 11, 13
Grammatik, LL(1) 13
Grammatik, LR(1) 13
Grammatik, mehrdeutige 12
Grammatik, *non-terminal symbol* 7
Grammatik, Rekursion 8
Grammatik, Startsymbol 8
Grammatik, *terminal symbol* 8
Grammatikbegriff 7, 11, 13
Gruppe von Anweisungen 138, 144, 157
Kommentar (in *lex*) 40

Kommentar, Muster 35, 37
Konfiguration 17
Konflikt (in *yacc*) 19, 20, 99
Konstante, Muster 35, 36
lex 33
lex und *yacc* 44, 51, 52, 54
lex, Bücherei 40
lex, Eingabesyntax 39
lex, Fehlersuche 44, 45, 48
lex, Kommentar 40
lex, Textersatz 49
lexikalische Analyse 31, 113, 120
linksrekursiv 9, 12, 146
LL(1) Grammatik 13
look-ahead 13
LR(1) Grammatik 13
main() 40, 42, 45, 56, 57, 119
mehrdeutige Grammatik 12
mehrdeutiges Muster 38
Mehrdeutigkeit, Vorschrift gegen 12, 21, 38, 82
message() 133, 202
Muster 33, 34
Muster für C Positionsangabe 45
Muster für Gleitkommakonstante 73
Muster für Kommentar 35, 37
Muster für Konstante 35, 36
Muster für Name 35
Muster für reservierte Worte 45
Muster für Zeichenkette 36
Muster für Zwischenraum 35
Muster, mehrdeutig 38
Muster, rechter Kontext 37
Musterwiederholung 37
Name, Muster 35
non-terminal symbol 7
Operator 31
Parameteradresse 145
Parameterdeklaration 114
Parser 51
Positionsangaben, C-Preprozessor 49, 61
Prozedursegment 139
Prozedursegment, Stack 139
push-down Automat 52
rechtsrekursiv 12, 147
reduce-Operation (in *yacc*) 18, 53
reduce/reduce Konflikt 20, 99
Regel 2, 7, 12
Rekursion 137, 138
Rekursion in Grammatik 8
rekursiv 9, 12, 146, 147
Repräsentierung eines Eingabesymbols 54
reserviertes Wort 31
reserviertes Wort, Muster 45
return 154, 164, 177
Satz 5
Semantik 5
Semantikfehler 109
shift-Operation (in *yacc*) 18, 19, 53, 70
shift/reduce Konflikt 19, 99
Simulation 169, 222
Speicherverwaltung 137, 213
Sprung, vorwärts 151, 154, 158, 178, 180, 181, 185
Stack bei Ausführung 137, 139, 164, 171, 174, 176, 177, 178
Stack für **break** 152, 154, 161, 184, 185
Stack für **continue** 152, 154, 161, 184, 185
Stack für Prozedursegment 139
Stack für Zustände (in *yacc*) 52
Stack-Maschine 52
Startsymbol 8
strsave() 78, 116, 203
Suche in Symboltabelle 113, 121
Symboltabelle 112, 137, 155, 187, 203
Symboltabelle, blindes Element 118
Symboltabelle, Suche 113, 121
Syntax 5
Syntaxanalyse 51
Syntaxbaum 8, 11, 12
Syntaxfehler 56, 235
Syntaxgraph 32
terminal symbol 8
Textersatz (in *lex*) 49
Transfervektor 185
Typ (eines Namens) 113
Typ für Wert-Stack (in *yacc*) 72, 134, 144

Typographie (in *yacc*) 28
Übergangsmatrix 31, 52
Variablendefinition 124
Vereinbarung 110, 121
Vorrang 12, 21, 24
Vorschrift 7, 12
Vorschrift gegen Mehrdeutigkeit 12, 21, 38, 82
Vorwärtssprung 151, 154, 158, 178, 180, 181, 185
warning() 133, 202
Wert-Stack (in *yacc*) 70, 122, 144, 155, 158, 165, 181, 183, 186
yacc 14, 51
yacc und *lex* 51, 52, 54
yacc, **%left** 21, 25, 54
yacc, **%nonassoc** 21, 54
yacc, **%prec** 82
yacc, **%right** 21, 25, 54
yacc, **%token** 15, 24, 45, 54, 55, 136
yacc, **%type** 136
yacc, **%union** 135
yacc, **accept**-Operation 52
yacc, Aktion 70
yacc, Aktion innerhalb einer Formulierung 83
yacc, anonyme Aktion 83
yacc, Bücherei 239
yacc, Eingabesyntax 21
yacc, **error**-Operation 52, 90
yacc, **error**-Symbol 89, 90
yacc, **error**-Symbol, Plazierung 93, 94
yacc, Fehlersuche 62, 239
yacc, **goto**-Operation 53, 70
yacc, Konflikt 19, 20, 99
yacc, **reduce**-Operation 18, 53
yacc, **shift**-Operation 18, 19, 53, 70
yacc, Stack für Zustände 52
yacc, Typ für Wert-Stack 72, 134, 144
yacc, Typographie 28
yacc, Wert-Stack 70, 122, 144, 155, 158, 165, 181, 183, 186
yacc, **yyerrok**-Aktion 93, 99
yacc, Zeigertyp für Wert-Stack 73
yaccpar Modifikation 235, 239
yyerrok-Aktion (in *yacc*) 93, 99
yyerror() 51, 62, 237
yymark() 61
yywhere() 60, 62
Zeichenkette, Muster 36
Zeiger, Speicherplatz 138
Zeigertyp für Wert-Stack (in *yacc*) 73
Zwischenraum, Muster 35

PC professionell
Arbeitsplatzrechner in Ausbildung und Praxis

Herausgeber: Prof. Dr.-Ing. Werner Heinzel, Fulda

In dieser Reihe sind bereits erschienen:

Banahan/Rutter, UNIX – Lernen, verstehen, anwenden

Feuer, Das C-Puzzle Buch

Fuchs, Einführung in BTX-Anwendungen

Gillner, Datenbanken auf Arbeitsplatzrechnern

Heinzel, Arbeitsplatzrechner

Kernighan/Ritchie, Programmieren in C

Meißner, Arbeitsplatzrechner im Verbund

Norton, MS-DOS und PC-DOS

Norton, Die verborgenen Möglichkeiten des IBM PC

Partosch, PASCAL mit Arbeitsplatzrechnern

Pomberger, Lilith und Modula-2

Schirmer, Die Programmiersprache C

Shoup, Numerische Verfahren für Arbeitsplatzrechner

Wolfe/Koelling, BASIC-Programme aus Naturwissenschaft und Technik

Carl Hanser Verlag, 8000 München 86

PC professionell
Arbeitsplatzrechner in Ausbildung und Praxis

Die vielseitigen und doch leicht verwendbaren UNIX-Programmgenerator-Werkzeuge „lex" (Textanalyse-Generator) und „yacc" (Parser-Generator) werden an Hand der Entwicklung eines Compilers im praktischen Einsatz vorgeführt. Darüber hinaus sind diese Programmierwerkzeuge vielseitig einsetzbar für viele Arten von Texterkennung und Textmanipulation.

Das Buch enthält den kompletten Programmtext des Compilers sowie eine Fülle von Hinweisen zum effizienten Einsatz der Generatoren. Es bietet eine Fallstudie über die systematische Entwicklung eines größeren Programmes und vermittelt die Beschreibung der prinzipiellen Komponenten eines einfachen Compilers.

Der Text ist ebenso als Einführung gedacht, die einem detaillierten Studium von Algorithmen zum Compilerbau vorausgehen sollte, wie auch als praxisbezogene Darstellung, wie man speziell die erwähnten Generatoren erfolgreich einsetzt. Der Leser sollte einen Texteditor bedienen und die üblichen Manipulationen mit Dateien ausführen können. Er muß in C programmieren können. Rudimentäre Kenntnisse anderer Programmiersprachen wie PASCAL, FORTRAN oder BASIC wären nützlich.

ISBN 3-446-14359-9

HANSER